DataGeneral

Software Documentation

Learning to Use Your Advanced Operating System (AOS)

069-000018-02

Learning to Use Your Advanced Operating System (AOS)

069-000018-02

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. 069-000018

Printed in the United States of America
Revision 02, December 1983

NOTICE

Learning to Use
Your
Advanced Operating System
(AOS)
069-000018

Revision History:

Original Release - June 1978

First Revision - November 1981

Second Revision - December 1983

Effective with:

AOS Rev. 3.4

CONTENT UNCHANGED

The content in this revision is unchanged from 069-000018-01. This revision changes only printing and binding details.

Preface

AOS is Data General's Advanced Operating System.

In an hour or two, you can develop a working sense of AOS by using it. This book leads you through the steps required to

- talk to the system through the Command Line Interpreter (CLI);
- use a text editor: either SED (a screen-oriented editor) or SPEED (a character-oriented editor);
- produce and run a FORTRAN program;
- produce and run a COBOL program;
- program in Extended BASIC; and
- write, produce, debug, and execute an assembly language program.

We don't describe all features of AOS, its CLI, other utility programs, or the compilers you'll be using. These are all detailed in the manuals described in *AOS Software Documentation Guide.*

If AOS is new to you, this book will give you a practical basis for using it. Later on, you'll use other books, such as the reference manual for your chosen language and the *Command Line Interpreter (CLI) User's Manual.*

To program in a language other than FORTRAN, COBOL, BASIC, or DG assembly, you'll use a different compiler; however, Chapters 1 and 2 will still be useful.

What Do You Want to Do?

This book assumes only that you are interested in learning about the Advanced Operating System (AOS).

If you want to write text -- programs, memos, letters, books, or other material -- this book shows you how to use a text editor. You need not be a programmer to take advantage of the computer system's text processing powers.

If you want to write programs, this book shows you how -- via a text editor and program examples in four languages. To understand the examples, you'll need some experience with the language involved.

How is the Book Organized?

Chapter 1	tells you about AOS.
Chapter 2	leads you through a session with AOS.
Chapter 3	is a reference chapter with common CLI commands.
Chapter 4	explains the screen-oriented text editor named SED. SED commands resemble ordinary English; the editor is easy to use.
Chapter 5	describes the character-oriented text editor named SPEED. SPEED is almost a programming language in itself. You can choose either the SED or SPEED editor. Both editors work with any text file.
Chapter 6	shows you how to produce and run a FORTRAN program.
Chapter 7	shows how to produce and run a COBOL program.
Chapter 8	shows how to produce and run a BASIC program.
Chapters 9 and 10	give some background for assembly language programming and show you how to produce, run, and debug an assembly language program.
Glossary	defines pertinent terms, like *batch* and *byte*. When you see a term you don't know, check the glossary.

How Do I Use the Book?

To learn the basics of AOS, read Chapter 1 and try the session in Chapter 2.

To simply use a text editor -- not build programs -- you'll read selected parts of Chapters 1 and 2 and all of Chapter 4.

To program, you may need a text editor, but will probably want to know the basic concepts of the system. Read Chapter 1 and try the session in Chapter 2. Then -- unless you want BASIC -- choose a text editor, read its chapter, and proceed to the language chapter of interest. For BASIC, you can skip the text editor chapter. There are five specific courses you can take, shown in Table P-1.

Whatever your interests, you can read -- and try out -- any chapters you like.

To *program* in a language other than FORTRAN, COBOL, BASIC, or DG assembly, you'll be using a different compiler but Chapters 1, 2, and optionally 3 will be useful nonetheless.

Of course, you can read *more than one* language chapter, and try more than one language example, if you wish.

What Other Books Will I Need?

You may want more information on the various subjects introduced in this manual. The following list includes other manuals you may be interested in.

Text Editors

Advanced Operating System/Virtual Storage (AOS/VS) SED Text Editor User's Manual, 093-000249
SPEED Text Editor (AOS and AOS/VS) User's Manual, 093-000197

Languages

A Guide To Using Business BASIC, 069-000028
Extended BASIC User's Manual, 093-000065
COBOL Reference Manual, 093-000223
FORTRAN IV User's Manual, 093-000053
FORTRAN 5 Reference Manual, 093-000085
FORTRAN 77 Reference Manual, 093-000162
SWAT™ User's Manual, 093-000258
AOS Macroassembler Reference Manual, 093-000192
AOS Debugger and File Editor User's Manual, 093-000195

Operating System

Command Line Interpreter (CLI) (AOS and AOS/VS) User's Manual, 093-000122
Introduction to the Advanced Operating System, 069-000016
AOS System Manager's Guide, 093-000193
AOS Programmer's Manual, 093-000120
AOS Link User's Manual, 093-000254

In addition the *AOS Console User's Handbook,* 093-000150, is a handy reference of all CLI commands. You may also be interested in the appropriate 14 series (hardware) manual for your ECLIPSE; it includes the instruction set and other pertinent information.

For a more complete listing of all AOS documentation, see the *AOS Software Documentation Guide,* 069-000020.

Table P-1. Courses to Take Through this Book

Text Editing	FORTRAN Programming	COBOL Programming	BASIC Programming	Assembly Programming
Chapter 1, section "How Do I Work with AOS?"	Chapter 1 Chapter 2 Chapter 4 or 5	Chapter 1 Chapter 2 Chapter 4 or 5	Chapter 1 Chapter 2 *Chapter 4 or 5*	Chapter 1 Chapter 2 Chapter 4 or 5
Chapter 2, through "Logging On"	Chapter 6	Chapter 7	Chapter 8	Chapters 9 and 10

Reader, Please Note:

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: $\left\{ \begin{matrix} \text{required}_1 \\ \text{required}_2 \end{matrix} \right\}$ 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.

Additionally, we use certain symbols in special ways:

Symbol	Means
⤶	Press the NEW LINE key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

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

In examples, we use

This typeface to show your entry⤶
This typeface for system queries and responses.

) is the Command Line Interpreter (CLI) prompt.

The examples in this book show both uppercase and lowercase characters. On an upper- and lowercase terminal, you can use all uppercase, all lowercase, or any combination you please.

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.
- If you experience software problems, please notify Data General Systems Engineering.

End of Preface

Contents

Chapter 1 - What is AOS?

Chapter 2 - Working at a Terminal

Chapter 3 - CLI Features and Commands

Chapter 4 - Writing Text with the SED Editor

Chapter 5 - Writing Text with the SPEED Editor

Chapter 6 - Instant FORTRAN Programming

Chapter 7 - Instant COBOL Programming

Chapter 8 - Extended BASIC Programming

Chapter 9 - Assembly Language Programming

Chapter 10 - Writing AOS Assembly Language Programs

Tables

Illustrations

Chapter 1
What is AOS?

This chapter explains

- AOS
- AOS compatibility with DG's AOS/VS
- How you develop programs with AOS
- Main memory
- What a file is
- Reading and writing records
- How to get programs from other systems into your AOS system

This chapter may contain terms that are new to you. You'll find definitions in the Glossary at the end of this manual.

AOS

AOS, the Advanced Operating System, is a general-purpose operating system that runs on Data General's ECLIPSE™ computers. AOS can be a

- timesharing system, in which many users work interactively with the system, each on his or her own terminal;

- batch system, in which the system runs jobs as streams and does not interact with user terminals;

- timesharing *and* batch system, which serves terminal users and also runs batch streams concurrently. This is a typical arrangement for AOS.

AOS can do all this because it is a multiprogramming system; it can run many programs simultaneously. Each program has its own share of system resources; and each is called a *process*. AOS creates a process for each terminal user and for each batch stream.

Each process is like a complete computer system; it has up to 64K bytes of main memory, it often has its own terminal, and it can use most system devices.

AOS can manage 64 concurrent processes of 64K bytes each by periodically swapping them between main memory and disk.

AOS supports multitasking. This means that each process can perform several different tasks concurrently, 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 it waits for a peripheral device (like mag tape, line printer, or disk) to complete an operation. Multitasking is further described in the *Introduction to the Advanced Operating System,* 069-000016 and in the *AOS Programmer's Manual,* 093-000120.

AOS is a secure system. As a user, you have unique access to your own process and files. Your programming mistakes cannot harm the system or other processes. The system manager or operator can parcel out privileges and system resources like main memory and disk space to all users, on whatever basis he/she chooses.

Is AOS Compatible with DG's AOS/VS?

DG's Advanced Operating System/Virtual System (AOS/VS), which also runs on ECLIPSE™ computers, is based on AOS. Nearly all AOS/VS programs work exactly the same way in both systems, but some languages differ. Programs written for AOS need only recompiling to run on AOS/VS.

After learning to use AOS, you will understand the basics for using AOS/VS -- and vice-versa.

How Do I Work with AOS?

You'll use a *terminal* (a device with a keyboard and television-like screen) to communicate with AOS.

Using the terminal, you'll log on by typing the username and password given by someone in authority. Then, the system will generally run a program called the Command Line Interpreter (CLI) on your terminal. You can type commands to the CLI, including commands that execute other programs like text editors or -- to build programs -- compilers.

When you're finished using the system, you'll log off by typing the command **BYE** to the CLI.

That's all there is to it.

How Do I Develop Programs with AOS?

Depending on your programming language interests, you'll use several system programs (called utilities in this book).

To program in FORTRAN, COBOL, or assembly language, you'll use the CLI extensively. CLI commands work the same way interactively or in batch: the CLI includes the functionality of JCL on non-interactive systems. Through the CLI, you will

1. use a text editor utility to write or edit each source program file;
2. use a compiler (or assembler) utility to compile the source file(s). Compilation produces an object file;
3. use the Link utility to build the object file into an executable program file;
4. execute the program file; and
5. if need be, debug the program file.

To program in BASIC, you may also need the CLI to execute BASIC; if so, you need only one command to activate BASIC. Some systems run BASIC automatically for each BASIC user who logs on; if this is true for you, you don't need the CLI at all.

Chapter 6 leads you through the steps to produce and run a sample FORTRAN program; Chapter 7 does the same thing with COBOL; Chapter 8 does it with BASIC; and Chapters 9 and 10 introduce you to and lead you through assembly-language programming.

Main Memory

AOS gives each interactive user and each batch stream up to 64K (65,536) bytes of memory to use as desired. This is enough for nearly all programs -- and far more than any program in this book requires.

But, if a program needs more memory, it can

- use overlays (which are segments of executable code that are brought into the same area one by one as needed);
- use shared routines (which are segments of commonly used code, placed in one part of memory and accessible to all user programs);
- divide into smaller programs that execute in order.

The COBOL compiler automatically sets up overlays for programs larger than 64K bytes -- effectively freeing COBOL programmers from concern with program size.

AOS system planners and managers have many options when they tailor systems for users. The *AOS System Manager's Guide,* 093-000193 and *AOS Programmer's Manual,* 093-000120 offer more information on this.

What is a File?

A *file* is a collection of information treated as a unit. This information can be discrete scraps of data, like April's sales figures or December's mortgage balances; it can be a list of phone numbers; it can be a text source file that forms the basis for a program; or it can be a program file: a series of computer instructions that does useful work, like computing interest.

Certain program files can read other files, act on what they read, and produce new, updated output files. Each utility Data General provides is a program file -- including the Command Line Interpreter Program (CLI) and AOS itself.

Files are stored on devices like magnetic disks and tapes. Disk files are the most versatile and the most common; and you'll be using them extensively. A disk file can be very large or very small; the largest file can store 4 billion bytes; the smallest, 0 bytes.

Filenames

You give a *filename* to each disk file to identify it. Filenames have from 1 to 31 of the following ASCII characters: upper- and lowercase letters, numbers, period (.), dollar sign ($), and underscore (_). For example,

```
trans_$income_received?may.10
```

is a valid (if cumbersome) filename.

In filenames, upper- and lowercase letters are treated the same; for example, the system sees no difference between filenames TEST and test .

A period and the last two or three characters in a filename often identify the contents of the file. For example, FORTRAN 77 source filenames usually end in .F77 .

Using the last few characters in this way is not required, but programming will be easier if you do it.

We recommend that, for the sake of simplicity, you use, and assume that AOS recognizes, the following conventions for filename endings (or suffixes):

- .SR for assembly language source filenames
- .CLI for CLI macro filenames (defined in Chapter 2 and in glossary)

- .F77 for FORTRAN 77 source filenames
- .FR for FORTRAN IV and FORTRAN 5 source filenames
- .CO or .CB for COBOL source filenames

In addition, when it compiles or assembles a source file, AOS replaces the conventional filename ending (or suffixes) in the source filename with .OB in the new object filename. And, when it builds the object file into an executable program file, Link replaces the characters .OB with .PR in the new program filename.

These are all the conventions you'll encounter in this book.

File Directories

After you've created a few files, you'll need some way to keep track of their names. You will probably want to group your files by category. AOS has a kind of file called a *directory* whose sole function is to contain other files.

You can create and use directories at will to help organize your files; for example, a directory named FORT could contain all FORTRAN files, a directory called COBOL COBOL files, and a directory called PERSONAL with appropriate sub or *inferior* directories for personal files.

You'll learn how to create and use directories in the next chapter.

Reading and Writing Records

This section gives you some background on AOS *record I/O* or record input/output. This is information that a program writes to and reads from devices like the terminal, line printer, magnetic tape, and -- most commonly -- disk files. If you're eager to start working at a terminal, you can ignore this section. But you may want to return to it later, as it provides some useful basics for future I/O.

Generally, a program must *open* a device before reading from it or writing to it. The open sets up a channel between the computer and device through which information can flow.

The terminal itself is an I/O device: the keyboard *inputs* information to the system; the screen or printer *outputs* information from the system. But in FORTRAN, COBOL, and BASIC, user programs always have access to the terminal via appropriate statements and they need not explictly open it. Assembly language programs must open it if they want to use it.

The line printer is an output device. DG FORTRANs have an internal connection called a *preconnection* that allows FORTRAN programs to write to the printer without opening it. Other programs must open the printer if they want to write to it.

Mag tape and disk files can be used for either input or output. A program that uses a tape or disk must open the file before it can write to it or read from it. When the program has finished its file I/O, it can close the file explicitly; or it can terminate. When a program terminates, the system closes its open files. The open, read, and write statements in FORTRAN, COBOL, and BASIC are straightforward and will be familiar to you if you know the language.

Record Formats

Generally, programs write and read information to and from files in groups called *records*. A record is simply a series of bytes (characters). Each write or read usually involves one record.

Each record's structure and length is determined by the program that writes it. Programs can write records one way and read them back another way; but usually, to make sense of the records, a program reads them in the same way they were written.

AOS offers four different *formats* for records. You can usually specify a format (or use the default format) when you open the file. The record formats are

Format	Definition
data-sensitive	a series of characters terminated by a delimiter character. The default delimiter characters are ASCII NEW LINE (12_8), shown as ↓ in this book, form feed (14_8), and null (000). If a program is writing or reading data-sensitive records, and it hits one of these characters, the system treats the characters preceding the delimiter as the record. All DG text editors and compilers require data-sensitive files.
fixed length	a series of characters whose length is constant. Usually the program specifies this length when it opens the file.
variable length	a series of characters whose length is determined by the write that creates the record. The system stores the number of characters *with* the record, so that, on a later read, it can pick up the entire record.
dynamic	a series of bytes whose length is determined by the items read or written. There is no inherent format for dynamic records; the program simply reads or writes them as a series of bytes.

Figure 1-1 shows how the different formats of records are stored.

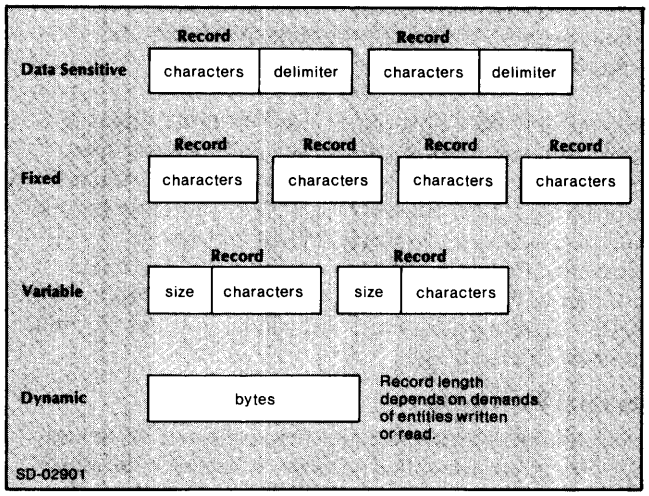


Figure 1-1. Record Formats

Data-sensitive records are most useful for writing and reading conversational (ASCII) text, because the default delimiters (NEW LINE, form feed, and null) occur as natural line terminators in ASCII text.

Fixed records are useful for information that adapts well to a constant length, like card images.

Variable records are quite flexible, but because the record size is stored with the record characters, the record may be difficult to interpret if you print it directly.

Dynamic records are read or written as a stream of bytes; the variables or data-item names in the program determine the number of bytes read or written. The system doesn't store them as discrete records. This means that you need to keep track of record length so that you don't read or write the wrong bytes.

For the programs shown in this book, you need not worry about record formats. But, for more sophisticated programs, you may want to set up the most efficient record format for the I/O you are doing. This is why we mention record format here.

How to Get Programs from Other Systems into Your AOS

This section explains how to transfer programs from other systems into yours. You may want to skip it on your first pass through this book. Read it when you want to transfer programs written on a different system into your AOS.

For Programs Written on a Data General System

We assume that the program sources have been copied to tape on Data General drives at the source site. The programs should have be copied via the DUMP command (not COPY or XFER).

For programs "dumped" from AOS or AOS/VS, you can simply use the LOAD command to load the tape files. First mount the tape on a local tape drive. Mounting a tape is described in Chapter 2; the LOAD command is described in Chapter 3.

For programs dumped from RDOS, DOS, or ICOS, mount the tape on a drive 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 drive 0, you could type

```
) XEQ  RDOS  LOAD/V  @MTB0:0  +/C
```

The parenthesis,), is the CLI prompt. The RDOS utility is further described in the *Command Line Interpreter (CLI) User's Manual.*

For Programs Written on a Different Manufacturer's System

Loading files will be easier if the files copied at the source site were copied onto an *unlabeled* tape. If they were copied onto a labeled tape, check the *CLI User's Manual* for the labeled tape command format.

Data General text editors and compilers require ASCII files with data-sensitive records. The standard Data General data-sensitive delimiters are ASCII NEW LINE (12_8 or $0C_{16}$), ASCII form feed (14_8 or $0E_{16}$), and and ASCII null (0). So -- if possible -- the source site files should be in ASCII, with NEW LINE as a record delimiter. If this is true, you can load the files with the COPY command (described in Chapter 3) and, generally, text edit them.

But if the file(s) are in EBCDIC, or their records have non-standard data-sensitive delimiters (like CR) or no delimiters (as for fixed records), you will need to convert the file(s). The easiest way to do this is with the Sort/Merge utility. Create a Sort command file via a DG text editor that tells Sort what to do; then execute Sort on the tape file.

For example, assume that the file is EBCDIC and contains fixed-length 80-character records in 50-record blocks (in IBM parlance, LRECL=80, BLKSIZ=4000). You might create a Sort command file named CONVERT with the following commands:

```
INPUT  FILE  IS  '@MTB0:0',
RECORDS  ARE  80  CHARS,
BLOCKS ARE 50 RECORDS.
OUTPUT FILE IS  'MY_CONVERTED_FILE'.
TRANSLATE  1/LAST  USING EBCDIC_TO_ASCII.
INSERT  '<012>'  AFTER LAST.
COPY.
END.
```

Then you'd execute Sort via:

```
) XEQ  SORT/C=CONVERT )
```

Sort would read from the first file (file 0) of the tape on drive @MTB0. It would produce file MY_CONVERTED_FILE, which you could then edit and whose program(s) you could compile. If the disk file contained more than one source program, you could use a text editor to copy each program from it into its own disk file.

Sort/Merge is further described in the *Sort/Merge User's Manual*, 093-000155.

Let's Get Started

For a "hands-on" session with AOS, proceed to Chapter 2.

End of Chapter

Chapter 2
Working at a Terminal

This chapter leads you through a sample session with your AOS system. It introduces many concepts, commands, and things like control (CTRL) characters. Descriptions of each command follow in Chapter 3.

The best way to learn AOS is to use it. So, for now, simply follow the steps we describe. The session will take between 30 minutes and an hour, but don't hurry.

AOS is a secure and forgiving system, with good error handling and explicit error messages. Any mistakes you may make will not harm the system or other users. So you can proceed with confidence.

Username and Password

You must have a *username* and *password* before you can use the system. These prevent unauthorized people from reading or changing authorized people's files.

If you already know your username and password, proceed. If not, find someone in authority (usually the *system manager)* and have him/her set you up with a username and password. For continuity in this book, we use the username and password JACK -- your username and password will obviously be different.

Finding a Terminal

Try to find a terminal with a cathode ray tube (CRT) display screen, resembling a small television screen. It should display a message like this:

system / TYPE NEW-LINE TO BEGIN LOGGING ON

If the only terminal available is a printing terminal, with the message printed on paper, then use the printing terminal; but be aware that using a printing terminal is more difficult (because it cannot erase characters) and slower.

If no terminal displays the message, see if the one you want is turned on. On CRT display terminals, the ON switch is to the lower right of the screen. Pull it gently outward; if it clicks, the terminal was off and is now on. Wait 10 seconds or so for it to warm up.

About the Keyboard

Assuming that you're using a DASHER ™ display terminal, look at its keyboard.

At the top, in a line, there are three groups or one group of unmarked keys. These *function keys* represent shorthand commands. We'll tell you about them later.

Below the function keys is the main keypad, which resembles that of a typewriter. Numbers 1 through 0 are at the top, letters below, and space bar at the bottom. There are some extra characters and some extra keys. The most important extra keys are ESC and CTRL, on the upper left, and NEW LINE, on the middle right. You'll be using ESC and CTRL later. But NEW LINE, on the right, tells the computer to take action; you'll be using it right away. The CR key next to NEW LINE usually -- but not always -- works the same way as NEW LINE.

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. Note that 0 and O, and and 1 and l, are *different characters* although they look somewhat alike in this typeset text.

To the right of the alphanumeric keypad, you'll see two other keypads. The numeric keys on the far right are simply those -- numeric keys. The pad between the main and numeric keypads includes the HOME key and may include a TAB key. You'll be using both of these later on.

Logging On

Having examined the keyboard, you're ready to log on by typing your username and password. To start, press

(NEW LINE key). The terminal will say

AOS n / EXEC n date time console-number
USERNAME:

It wants a username. Type in your username, followed by NEW LINE ( ). Take your time; you have 30 seconds to respond. For example, type

```
                JACK ↓
PASSWORD:
```

Now it wants your password, so type in the password and ↓; for example, JACK↓. The system won't echo (show) the password, so you can't see whether you made a mistake. But its okay if you do. When you make a mistake, or your username and password are not what you typed, the system will say

INVALID USERNAME-PASSWORD PAIR
USERNAME:

If you get this *INVALID* message, enter your username and password again.

When you have entered the correct username and password, the terminal will say

....messages....
LAST PREVIOUS LOGON date time

AOS CLI REV n date time
)

There may be no *messages*, or there may be a lot, depending on what the system manager wants to say to users.

The *CLI* is the Command Line Interpreter, a system utility program that gives you access to all other programs. The CLI prompt is a right parentheses,) . You'll see it hundreds of times in the future. (As mentioned in the Preface, this book shows all user input *and standard prompts* [like)] in THIS TYPEFACE; it shows all system output aside from standard prompts in *THIS TYPEFACE*. A □ is used to *emphasize* a space where it might not be obvious.)

(If you make several mistakes entering your username and password, the system will say *TOO MANY ATTEMPTS, CONSOLE LOCKING FOR 10 SECONDS*. No harm done. Wait 10 seconds and start again. Or, if you are dialing in to a remote AOS system, hang up, redial, and start again.)

When you see the) prompt, you are logged on to AOS. To log off, you can type the CLI command BYE↓ at any time. The log on/log off procedure will soon become second nature to you.

If you are interested *only* in text editing -- not the CLI -- go to Chapter 4.

For a hands-on session that will teach you about the CLI, read on.

Upper- or Lowercase?

If your terminal has both uppercase and lowercase characters, you can use either case at will. For example, either JACK or Jack or jack would have allowed you to log on. The system translates lowercase to uppercase internally, so jack is really JACK, but you never see this. Certain compilers, like FORTRAN IV and COBOL, require uppercase letters, but we mention this wherever it pertains. With the CLI, you can use whatever case you like.

Starting the Logfile

To get started, type

```
) LOGFILE  LOG.FILE ↓
)
```

This creates a file named LOG.FILE to record all terminal dialog. It will grow during the session and you can examine it afterwards, if you want.

LOGFILE is a CLI *command;* LOG.FILE is an *argument* to the command. Between commands and arguments, you must use at least one space. You can use more than one space, one or more tabs, or combinations as desired.

Directories and Files

As a typical user, you have your own *user directory,* bearing the username you logged on with. Your user directory is the exclusive property of your username -- people who don't use your username and password can't use it.

Type

```
) DIR ↓
:UDD:JACK
)
```

The DIRECTORY command (abbreviated DIR) tells you the *working directory* name. The working directory is simply the directory where you are. When you log on, the working directory is your user directory. (As always in this chapter, your own username will appear instead of JACK.)

:UDD:JACK describes the *path* to the working directory, JACK. It's a *pathname.* JACK is at the end of a path though other directories. From the top, the path looks like Figure 2-1.

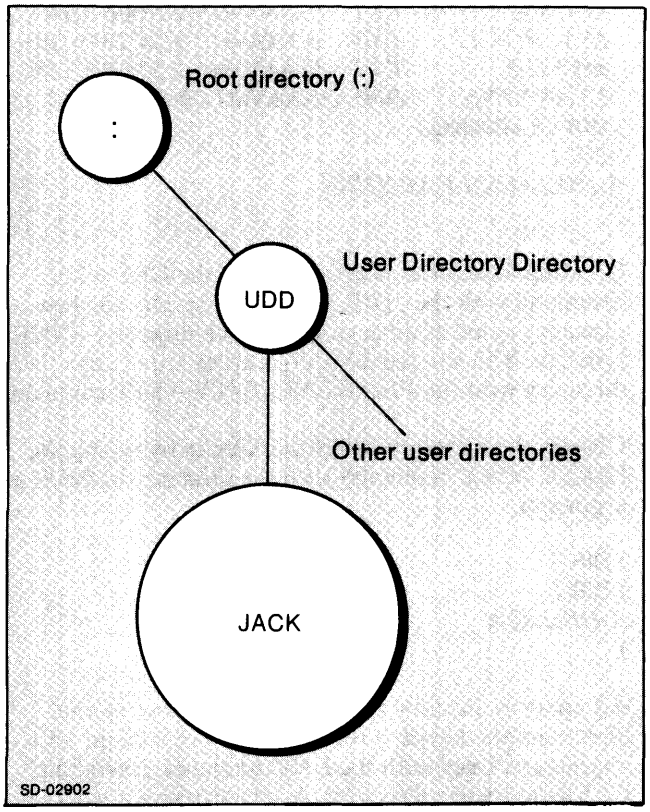


Figure 2-1. Path to a User Directory

But directories are practically useless without files. So, type

```
) CREATE   MYFILE ⤸
) Files   MYFILE ⤸
DIRECTORY    :UDD:JACK

MYFILE
)
```

The CREATE command creates a file; here, we've created a file named MYFILE.

FILESTATUS is one of the most useful commands. You may use the abbreviation FILES or F. In the last example, FILESTATUS described the working directory (JACK) and verified the existence of MYFILE.

Type

```
) FILES/AS   MYFILE ⤸
DIRECTORY    :UDD:JACK
MYFILE              TXT     9-JAN-81    9:46:50       0
)
```

The /AS appended to FILES is a *switch*. Switches change the meaning of a command; they may also change the operation performed on a command's argument(s). *Command switches* apply to commands, and *argument switches* apply to arguments. The /AS command switch tells the CLI to display an /assortment of information about the file.

The /AS assortment includes

- the type of file (TXT)
- the date the file was created or last modified (9-JAN-81)
- the time the file was created or last modified (9:46:50)
- the length of that file in bytes (0)

A *byte* is 8 bits or one character; in this case, MYFILE is empty. Obviously, the date and time will differ for your own MYFILE. Type

```
) DELETE/V   MYFILE ⤸
DELETED    MYFILE
) FILES/AS   MYFILE ⤸
)
```

DELETE (abbreviated DEL) does pretty much what you'd expect: removes the file. The /V switch tells the CLI to verify the deletion, which it did in the example by saying *DELETED MYFILE*. The FILES/AS command displays nothing, which means that the file was, in fact, deleted.

```
) CREATE/I   MYFILE ⤸
)) Greetings from MYFILE. ⤸
)) )⤸
  -- ----- you type this parenthesis and ⤸.
)
```

CREATE with the /I switch tells the CLI to create the file, take text from the terminal, and insert this text in the file. During the insertion, the CLI types 2 prompts [ ))] at the beginning of each NEW LINE. To end the Insertion, type your own right parenthesis and NEW LINE [ )] -- as above. CREATE/I can be useful for writing little programs and CLI macros. Type

```
) FILES/AS   MYFILE ⤸
DIRECTORY    :UDD:JACK
MYFILE              TXT     9-JAN-81    9:52:50       24
) TYPE   MYFILE ⤸
Greetings from MYFILE.
)
```

The new MYFILE is type TXT and has 24 bytes (characters). (Your MYFILE might have more or less, depending on the number of spaces you inserted.) The TYPE command (abbreviated TY) displays the contents of a file on the terminal screen.

```
) CRE/I   MACRO.CLI ⤸
)) WRITE□MYFILE□contains□;TYPE□MYFILE ⤸
)) )⤸
  -- ------ you type this parenthesis and ⤸.
)
```

One of the great features of the CLI is that it allows you to create your own commands, called CLI *macros*. If a filename consists of name and .CLI, when you type the name↵, the CLI will try to execute all commands in the file. Try it:

```
) MACRO ↵
MYFILE contains
  Greetings from MYFILE.
)
```

You wanted MACRO.CLI to write "MYFILE contains", and then type MYFILE; and you succeeded. The WRITE command simply writes the characters that follow it; it is useful within macros. The semicolon between the WRITE and TYPE commands delimits them; it allows you to write more than one command on a line (e.g., FILES MYFILE;TYPE MYFILE↵).

Now, having created two files, let's check them:

```
) FIELS/AS ↵
ERROR: NOT A COMMAND OR MACRO, FIELS
FIELS/AS
)
```

Typos cause error messages -- a good reason for using abbreviations. Let's try it again:

```
) F/AS ↵
DIRECTORY  :UDD:JACK

 LOG.FILE     TXT  9-JAN-81  9:51:06   482
 MYFILE       TXT  9-JAN-81  9:52:50    24
 MACRO.CLI    TXT  9-JAN-81  9:54:22    35

)
```

F (ILESTATUS) *without* an argument describes all files in the working directory. There may be files other than MACRO.CLI, MYFILE and LOG.FILE. Possibly there will be a .CLI macro that you didn't write. This macro was supplied by the system manager to set certain conditions for you as a user. Later, if you want, you can edit this macro to change the conditions and -- perhaps -- display a personal log-on message.

A New Directory

Now let's create a new directory. Directories help you keep track of different files; for example, FORTRAN programs could be in one directory, COBOL in another, BASIC in another, personal letters in another, and so on.

```
) CRE/DIR LEARNING ↵
) F/AS ↵
DIRECTORY  :UDD:JACK

 LOG.FILE     TXT  9-JAN-81  9:51:06   647
 MACRO.CLI    TXT  9-JAN-81  9:54:22    35
 MYFILE       TXT  9-JAN-81  9:52:50    24
 LEARNING     DIR  9-JAN-81  9:56:44     0
) DIR LEARNING ↵
) DIR ↵
:UDD:JACK:LEARNING
)
```

Creating directories is easy: you use the CREATE command with the /DIR switch and specify the new directory name as an argument. Each directory is DIR type file. You can see what directories are in any directory with the FILESTATUS/TY=DIR command.

Change the directory you're working in by typing the DIRECTORY command *with* the directory name as an argument.

```
) DIR ↑
) DIR ↵
:UDD:JACK
)
```

An uparrow (usually the SHIFT and 6 keys on the alphanumeric keypad) is one of many shorthand CLI characters. Used with the DIR command, it tells the CLI to go *up* one directory. As you can see, it worked; the working directory is now the original user directory, not LEARNING.

```
) MOVE/V  LEARNING  macro.cli myfile ↵
   MACRO.CLI
   MYFILE
)
```

MOVE, another handy command, copies files into another directory. The /V switch told the CLI to verify the MOVE, which it did. The first argument to MOVE must be the name of the directory, LEARNING, to which you will move files. The following arguments are the filename(s) you want moved.

The original files are still in directory JACK; copies are in directory LEARNING.

Screen Editing

Now let's try a little screen editing. (Skip this section if you're working on a printing terminal.) With the ring finger (or pinky) on your left hand, press the CTRL key and hold it down. While you're holding CTRL down, press the A key. This is called CTRL-A. Release both keys. On your screen, you will see the last command line repeated, just as you originally typed it:

```
) MOVE/V  LEARNING  macro.cli  myfile
```

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 myfile. Press NEW LINE:

```
                    )
WARNING: FILE ... EXISTS: LEARNING:macro.cli
WARNING: FILE ... EXISTS: LEARNING:myfile
)
```

By pressing), you re-entered the MOVE command, which provoked an error message because the files already existed in directory LEARNING. The error is okay: it was deliberate and didn't harm anything. Enter CTRL-A again, and the MOVE command will appear once more:

```
) MOVE/V     LEARNING     macro.cli     myfile
```

Now let's edit the command line on the screen. The cursor is at the end of the line. Press the HOME key to the right of the main keyboard (if there is no HOME key, press CTRL-H). HOME or CTRL-H moves the cursor to the beginning of the line, after the prompt.

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

```
) DELETE/VEARNING     macro.cli     myfile
```

The cursor will be ahead of the last character typed, in this case on the E after the V. Press the space bar until you've wiped out the rest of EARNING. The command line should look like this:

```
) DELETE/V                    macro.cli     myfile
```

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

After the cursor reaches the end of the line, press). This tells the CLI to execute the DELETE command:

```
                              )
DELETED   macro.cli
DELETED   myfile
)
```

Thus you have used screen editing to change the MOVE command to a DELETE command. This DELETE just eliminated duplication, deleting the files from your user directory. Copies of the files are safe in directory LEARNING.

For practice, try it again. Enter CTRL-A to redisplay the DELETE line. Then press HOME to move the cursor to the beginning of the line. Finally, press CR to eliminate the command line.

The major screen editing characters are

CTRL-A	redisplays the last command typed.
CTRL-F	moves cursor forward to first character of next word.
CTRL-B	moves cursor backward to last character of previous word.
→ key or CTRL-X	moves forward one character
← key or CTRL-Y	moves backward one character.
CR or ERASE EOL key	deletes all characters to the right of the cursor. CR also tells the CLI to execute the command.
CTRL-E	inserts new text or terminates an insert.
DEL key	deletes character to left of cursor.

Although describing it required many steps, screen editing is really very easy and fast. It can save you a lot of aggravation, when, as will often happen, you make a typing mistake or want to re-enter a long command line. (The CTRL key by itself does nothing, but in conjunction with other characters -- like A, E, or F -- it can do a lot.)

These CTRL (and DEL) characters work the same way with the SED text editor as with the CLI -- so you may be using them often. Don't be afraid to experiment with them.

Filenames and Pathnames

At this point, we're in the user directory, having just moved our files into directory LEARNING. Let's try to type one of the files:

```
) TY  MYFILE )
WARNING: FILE DOES NOT EXIST,... MYFILE
)
```

Try a *pathname* that includes directory LEARNING:

```
) TY  LEARNING:MYFILE )
Greetings from MYFILE.
)
```

A pathname specifies a path to the file and always includes the file's name. Generally, if a file is not in the working directory, the pathname must include one or more directory names.

If the file is in an *inferior* (lower) directory, you need specify only the inferior directory name(s) in the pathname -- as above.

If the file is in the working directory, the filename can be the pathname:

```
) DIR  LEARNING ⤸
) TYPE  MYFILE ⤸
Greetings from MYFILE.
)
```

If the file is in a *superior* (higher) directory, you must specify a pathname that includes the directory:

```
) TY LOG.FILE ⤸
WARNING: FILE DOES NOT EXIST,... LOG.FILE

) TY ↑LOG.FILE ⤸
.
(text of LOG.FILE)
.
)
```

A file's full pathname always works, but can be laborious to type -- so use the ↑ character as needed. In most cases, as you work with AOS, you'll find it easier to use the DIR command to get into the directory you want instead of using long pathnames.

Legal Filenames and Pathnames

A filename can contain any of the alpha characters A-Z, numbers 0-9, underscore (_), period (.), question mark (?) and dollar sign ($). It can be up to 31 characters long. This allows you to use explicit filenames. For example, you could name a file

```
AOS_1ST_TIME.DRAFT1
```

Each pathname consists of one or more legal filenames; if a pathname contains more than one filename, separate all names with colons. A space is a delimiter and cannot be part of a filename or pathname.

Keeping Track of Filenames

You may not have many files now, but eventually you'll have a lot. Directories can help classify them, but even within a directory you may have hundreds of files.

For an alphabetical list of filenames, use FILESTATUS with the /SORT switch. Type

```
) F/AS/S ⤸
DIRECTORY  :UDD:JACK:LEARNING

 MACRO.CLI   TXT  9-JAN-81  9:54:22    35
 MYFILE      TXT  9-JAN-81  9:52:50    24
)
```

Directory LEARNING doesn't provide much of a challenge for F/S, but the example makes the point.

Filename Templates

Templates specify a set of filenames. The most common template characters are

Char.	What it Means
*	Match any single character except a period.
-	Match any series of characters not containing a period.
+	Match any series of characters.
\	*Omit* a series of characters.
#	Search the specified directory *and all inferior directories*. Without this template, the search is restricted to the working (or specified) directory.

To see all filenames that are six characters long, and that don't contain a period, type

```
) FILES  ****** ⤸
DIRECTORY   :UDD:JACK:LEARNING

MYFILE
)
```

To see *all* filenames that don't contain a period, type

```
) FILES  - ⤸
DIRECTORY   :UDD:JACK:LEARNING

MYFILE
)
```

Or to see all filenames whose names end in .CLI, type

```
) FILES  -.CLI ⤸
DIRECTORY   :UDD:JACK:LEARNING

MACRO.CLI
)
```

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

```
) FILES  M+ ⤸
DIRECTORY   :UDD:JACK:LEARNING

MYFILE          MACRO.CLI
)
```

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

```
) FILES   +\+.CLI ↓
DIRECTORY   :UDD:JACK:LEARNING

MYFILE
)
```

The # character allows you to search multiple directories. For example, type

```
) FILES   :UDD:JACK:# ↓

DIRECTORY   :UDD

JACK
DIRECTORY   :UDD:JACK

LOG.FILE
DIRECTORY   :UDD:JACK:LEARNING

MYFILE              MACRO.CLI
)
```

To search all directories for a file named FOO, type

```
) F/AS   :UDD:JACK:#:FOO ↓

DIRECTORY   :UDD:JACK

DIRECTORY   UDD:JACK:LEARNING

)
```

You can use the switches /AS and /S with the template characters.

For a hypothetical example, assume that you have a lot of FORTRAN 77 source files, which conventionally end in .F77 . You want to see and sort all their names. You'd type

```
) FILES/AS/S   +.F77
```

The resulting listing might be

```
ARRAY_1.F77     UDF    10-JAN-81  10:46:50   9562
BIORHYTHM.F77UDF       12-JAN-81   9:01:56   2337
BOOMER.F77      UDF    10-JAN-81  14:48:01   8334
MORTGAGE.F77 UDF        9-JAN-81  11:01:33   2023
OPEN_TEST.F77 UDF       9-JAN-81  13:14:02   1356
```

Templates can also help you to organize, dump, or delete your files selectively.

The CLI commands that allow template characters include FILESTATUS, DELETE, TYPE, MOVE, PERMANENCE, DUMP, and LOAD (PERMANENCE, DUMP, and LOAD are described later). If you ever want to use a template and are not sure that your command accepts it, just try the template.

But be careful with the DELETE command and templates: if you accidentally insert a space after a template character (e.g., DELETE□+□.ST↓) you might delete all files in the directory.

Printing Files

To print one or more files, use the QPRINT command, which can be abbreviated QPR. Type

```
) QPR   MACRO.CLI ↓
QUEUED ,   SEQ=n,  QPRI=n
)
```

QPRINT sends the text of a file to the line-printer queue. The line printer needs a queue because it is so much in demand.

Go to the printer and it will handle your request soon, if not immediately.

Handling the Line Printer

At the printer, check with the system operator (if there is one) and find out who is supposed to work the printer controls. There may be a form feed macro that outputs blank pages between files printed so that no one needs to work the controls.

- If there is a form feed macro, go back to your terminal and type its name; then return to the printer and get your file. Use the macro in the future after you print a file.
- If the system operator is supposed to work the printer controls, let him/her do it and hand you the printed 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. Press ON/OFF LINE to put the printer back on line. Tear off your printed file.

We tell you all this because the way the paper folds as it emerges from the printer is important to its readability -- so, if you work the controls, *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. Also, be sure to put the printer back ON LINE so it can continue serving users.

Each printing job prints a header that lists username, pathname, and time; the job follows the heading with the printed text of the file(s) specified. In this case, you'll see the text of MACRO.CLI.

In many commands -- including FILESTATUS -- you can apply the switch /L=@LPT . This switch sends output from the command to the printer instead of the terminal. It is very handy for things like file sorts and program listings from compilers. For example:

```
) FILES/AS/S/L=@LPT ↵        (Output to printer.)
)
```

Taking a Break

At this point, you've learned about logging on; about the LOGFILE, DIRECTORY, CREATE, FILESTATUS, DELETE, TYPE, WRITE, and QPRINT command; and about switches.

You've CREATEd a directory, used MOVE, tried screen editing, learned something about filenames, pathnames, and templates, and printed a file.

Your directory and your file structure look like Figure 2-2.

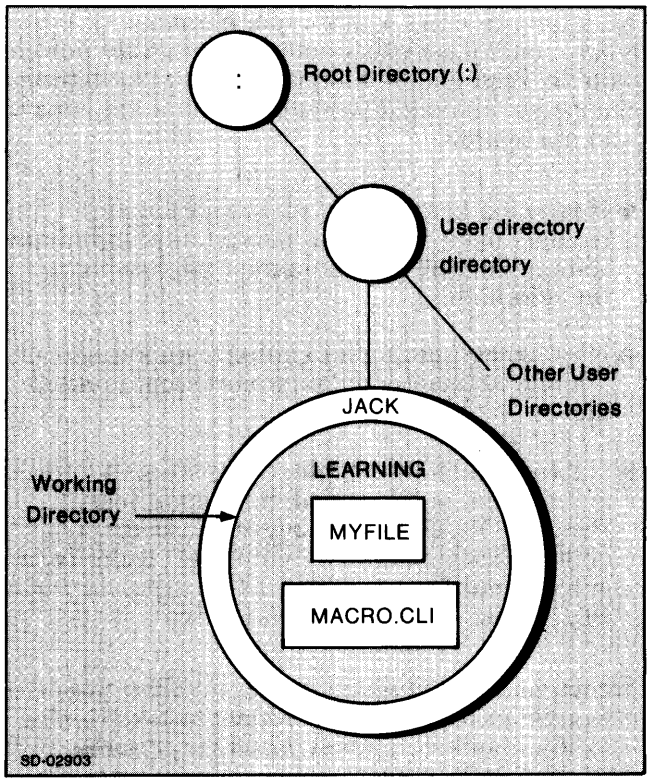


Figure 2-2. In Working Directory LEARNING

If you want to log off, type BYE↵ and the CLI will terminate your user process and log you off. Later, when you log on again, type LOGF LOG.FILE↵ and DIR LEARNING↵.

To proceed, read on.

Utilities and Searchlists

If you wanted to work exclusively in the CLI, you wouldn't need this section. But much -- if not most -- of your work will involve system utilities: text editors, compilers, and the Link program.

Putting copies of each utility in every user directory would consume a lot of storage space -- so there is only one copy of each, in a directory called :UTIL.

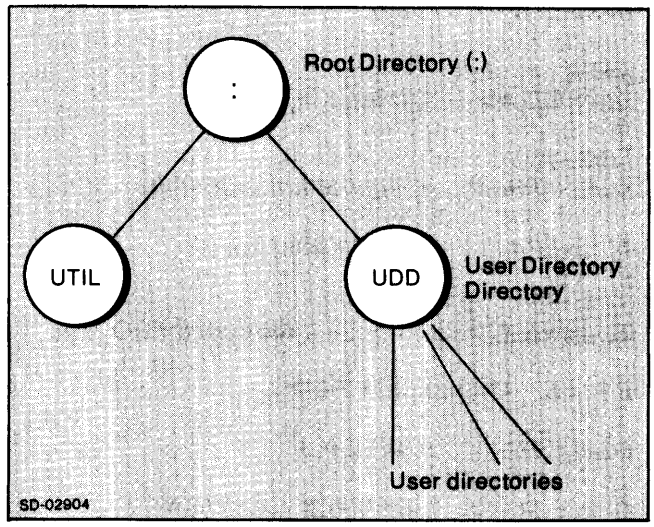


Figure 2-3. Path to Directory :UTIL

Users can reach any program in :UTIL via something called 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. Type

```
) SEARCH ↵
  ..., :UTIL, ...
)
```

The SEARCHLIST command without an argument displays the current searchlist; with an argument, you can use it to change the searchlist. Your own searchlist may include directories other than :UTIL, but practically all searchlists include :UTIL. The system manager usually provides a log-on macro that contains a SEARCHLIST command; this establishes the default search list.

Type

```
) XEQ SED FOO ↵
Do you want FOO to be created? NO ↵
Start over? NO ↵
)
```

You just executed the SED text editor program, detailed in Chapter 4. (If you got an error message from XEQ, just type the word SED).

When you answered NO⤶ twice, the editor terminated and returned to the CLI. Now change the searchlist to *omit* :UTIL and try executing the editor again:

```
) SEARCH  :UDD:JACK:LEARNING ⤶
) SEA ⤶
:UDD:JACK:LEARNING
) XEQ SED FOO ⤶
ERROR:  FILE DOES NOT EXIST
XEQ,SED,FOO
)
```

So, restore the old searchlist, but add something to it:

```
) SEA  :UTIL,  :UDD:JACK:LEARNING ⤶
) SEA ⤶
:UTIL, :UDD:JACK:LEARNING
) XEQ SED FOO ⤶
Do you want FOO to be created?  NO ⤶
Start over?  NO⤶
)
```

With :UTIL in the searchlist, you again have access to the editor and other utilities. By adding :UDD:JACK:LEARNING to the search list, you provided access to any file in LEARNING without using a pathname specifier. Note that if your search list contains one or more inferior directories, you should not use the same filename for files in different directories.

If ever you get *DOES NOT EXIST* or obscure error messages when you try to execute a program that you think exists, check your search list. It may be wrong -- but, as you can see, it's easy to fix.

System Control Sequences

There are several system CTRL sequences that you will find very useful (aside from the screen editing CTRL characters shown earlier). These CTRL sequences do such things as suspend and resume display, clear the screen, interrupt execution of a CLI command, and abort the current process. The most common CTRL sequences are

CTRL Sequence	What it Does
CTRL-L	Clears the screen and executes command next to) prompt (if any).
CTRL-S	Postpones display. To display all suspended information, enter CTRL-Q.
CTRL-Q	Resumes display suspended by CTRL-S.
CTRL-O	Cancels display. To resume display, enter CTRL-O again.
CTRL-U	Erases the current command line.
CTRL-C CTRL-A	Interrupts execution of the current command (in CLI, SPEED and SED text editors, and BASIC only).
CTRL-C CTRL-B	Aborts the current program, returns 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, using a big parameter file called PARU.SR in directory :UTIL. Start by entering

CTRL-L (Hold down CTRL key and press L.)

to clear the screen. Then type

```
) TYP :UTIL:PARU.SR ⤶
.....
; COPYRIGHT (C) DATA GENERAL
CORPORATION...
; ALL RIGHTS RESERVED...
.....(text of PARU.SR).....
```

CTRL-S

As you see, display has stopped. 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.
```

CTRL-S

Use CTRL-S and CTRL-Q to break a long file into readable segments. But here, we don't want to read the rest of PARU.SR, nor do we want to wait the several minutes needed to type it all on our terminal. What do we do? Enter

CTRL-C CTRL-A

CTRL-C CTRL-A should interrupt the TYPE command, but it appeared to do nothing. It *did,* in fact, interrupt the TYPE command but we can't tell because display is still suspended with CTRL-S. So enter

CTRL-Q

```
ERROR: CONSOLE INTERRUPT
)
```

A *CONSOLE INTERRUPT* isn't really an error, but the CLI had to say something about it. Now do it again. Enter CTRL-A to make the TYPE command reappear, press ↓, and alternately press CTRL-S and CTRL-Q to freeze and unfreeze the display. Finally, when you're bored by PARU, enter CTRL-C CTRL-A again.

From this, you'll note that when CTRL-S is in effect, the system appears to have died. It hasn't -- all you need do is enter CTRL-Q. (CTRL-O gives the same visual effect as CTRL-S, but actually throws away the output; it can help speed up programs that do a lot of writing to the screen, because the writes slow down the programs. You'll rarely want CTRL-O when you're in the CLI. Still, if the screen appears frozen and CTRL-Q has no effect, try CTRL-O.)

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. Type

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

So, having tried CTRL-L, CTRL-S/CTRL-Q, CTRL-C CTRL-A, and CTRL-U, all that remains is CTRL-C CTRL-B. Don't type this now; it will log you off the system. To try it, type

```
) XEQ SED ↓
Name of file to edit?    CTRL-C CTRL-B ↑C↑B

*ABORT*
CONSOLE INTERRUPT
ERROR: FROM PROGRAM
XEQ,SED
)
```

As you can see, CTRL-C CTRL-B has a mighty effect. It's most useful during compilations when you know that there are mistakes in the source and you don't want to wait for the compiler to finish. Be careful with it otherwise: if you enter it from a text editor, all your edits will vanish at the abort; if you issue it from the CLI, your user process ends and you'll have to log on again.

File Access Control

File access control governs user access to files within the system. For an example, type

```
) DIR  : ↓
) F/AS ↓
  WARNING: FILE ACCESS DENIED, FILE=
) DIR  :UDD ↓
) F/AS ↓
  WARNING: FILE ACCESS DENIED, FILE=
) DIR/I  LEARNING ↓
) DIR ↓
  :UDD:JACK:LEARNING
)
```

As you saw, you are denied access (even to see the file names) in the root directory and :UDD. DIR/I specifies your initial (user) directory and LEARNING takes you back to LEARNING. DIR/I is useful when you want to get back to a home directory quickly.

Type

```
) ACL/V myfile ↓
  myfile          JACK,OWARE
)
```

Each file in the system has an *Access Control List* (or ACL). When a user creates a file, as you did MYFILE, the system gives that file an ACL with all privileges *(OWARE)* for that user. On some systems, all users (+, as in filename templates) may automatically be given privileges RE. ACLs for system files do not give all users *any* privileges, which is why you couldn't even list the files in the root and :UDD. The privileges -- O, W, A, R, and E -- have the following meanings:

	Privilege	**Allows**
O	Owner	If a user has Owner access to a directory, he/she can change its ACL -- giving him/herself other privileges at will. Owner access to a directory also allows a user to delete or change the ACLs of *files* in the directory.
W	Write	A user with Write access to a directory can create and delete files, and change file ACLs. With Write access to a file, a user can modify the file's contents.
A	Append	With Append access to a directory, a user can add files to it. Append does not apply to files.
R	Read	With Read access to a file, a user can read the file (e.g., with TYPE) providing he/she has at least R and E access to the directory.
E	Execute	E applies primarily to to programs like system utilities and those you may create. For a directory, E allows a user to execute programs within the directory providing he/she also has at least E access to the program file(s). E does not allow a user to read filenames in a directory or read the text of a file. A user with Read and Execute access to a directory can DIR into the directory, use the directory name in a pathname, and read the filenames in the directory (e.g., with FILESTATUS). R and E access privileges are often given together.

In your own directory, with Owner access to everything, you can change file and directory ACLs at will -- despite their apparent ACLs.

Generally, selective ACL settings are most useful in directories that users share, for example common directories of macros or games. If such a directory has +,RAE access, and its superior directories have at least +,E access, you can MOVE files there for general use; and you can read files and execute programs there.

Note that whenever a file is moved or dumped, by default it retains its original ACL. So if you receive a file from someone else, you may get curious error messages from text editors or other utilities when they try to use the file. This often occurs because the ACL doesn't give the utility access to the file. In such cases, check the file's ACL and give yourself OWARE privileges to it if you don't have them. You can do this globally in any of your directories by typing

```
) ACL  +  your-username, OWARE ↓
```

Making Files Permanent

If you are ever careless using a template, you might accidentally delete a lot of cherished files, or even directories. For example, type

```
) CREATE/DIR FOO ↓
) DIR FOO ↓
) CREATE FOOFIL ↓
) DIR ↑
) F/AS FOO ↓

DIRECTORY :UDD:JACK:LEARNING

FOO    DIR    9-JAN-81  10:12:48  4608
) DEL/V FOO ↓
DELETED  FOO
)
```

It's as easy as DEL/V to delete a directory or file -- but not if you turn PERMANENCE on for it. PERMANENCE makes the file or directory permanent; no one can delete it until you turn PERMANENCE OFF. Try it:

```
) CRE/DIR  FOO ↓
) PERM  FOO  ON ↓
) PERM/V  FOO ↓
FOO     ON
) DEL/V  FOO ↓
WARNING:CANNOT DELETE PERMANENT FILE

) PERM  FOO  OFF ↓
) DEL/V  FOO ↓
DELETED  FOO
)
```

As you see, PERMANENCE is an easy way to prevent deleting your files. The command accepts templates, so you can set it easily for all your files; e.g., via PERM + ON↓.

But, PERMANENCE can interfere with everyday program development because text editors and other utilities internally delete files and recreate them. So you may want to set PERMANENCE on only for directories and selected files. PERMANENCE will not save a file if its parent directory is nonpermanent and you delete the directory.

Backup for Your Files

At some point, you will want to know how to make backup copies of your files (copies for safekeeping). If you don't want to do this just now, you can skip this section and go on to "Getting Help".

Most systems have a procedure for backing up user files. Often, the backup medium is magnetic tape. But tape drives are useful and valuable devices, and users may not be allowed to operate them on your system. (The amount of access users have to devices depends on your system: the operator may run *all* devices, or run tape drives only; or there may be no operator, in which case you will be operating both the line printer and tape drives yourself.)

In any case -- although backup of your files may be done for you -- it's useful to know how to do it.

The traditional way to get a tape mounted is with the MOUNT command, which sends a MOUNT request to the operator's terminal. Type

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

Now, one of four things will happen:

- You get the message *ERROR: OPERATOR NOT AVAILABLE.* If this occurs, skip to the section "If an Operator Doesn't Respond", below.

- You get 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 him/her about procedures. Having talked with the operator, you may want to try again later, or skip all the way to "Getting Help", below.

- After a while (say 30 to 60 seconds), you get the CLI prompt,), back from 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 drive is a *linkname,* determined by the first argument to the MOUNT command. In this case, because you typed MOUNT TAPE, the tape linkname 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 tell the system he/she is not on duty. If this command had been issued, you'd have gotten the *OPERATOR NOT AVAILABLE* message. 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, then type DISMOUNT TAPE↵ and skip to the section "If an Operator Doesn't Respond", below.

If an Operator Responds

If the CLI prompt,) returns from your MOUNT command, type

```
) DIR/I ↵
) DUMP/L=@LPT   TAPE:0 ↵
```

The DUMP command copies (dumps) disk files to a destination file. If you type DUMP from your user directory and omit file (path) names, DUMP dumps *all* files from the user directory and inferior directories. In this case, the dump copies to tape file 0 (the linkname for TAPE file 0 is TAPE:0), but you could have specified a disk file. DUMP files can, and usually do, contain more than one disk file. The /L=@LPT switch directs a listing of all dumped files to 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 asks the operator to dismount the tape. Go to "After Dumping to Tape", below.

If an Operator Doesn't Respond

If you got an *OPERATOR NOT AVAILABLE* message or no answer from MOUNT, you'll need to mount the tape and operate the drive yourself. Follow these steps:

- Get a reel of magnetic tape that has a yellow plastic ring in it. This *write-enable* ring allows you to write on the tape.

- Take the tape to the system's tape drive. If there are several drives, try to find an empty one. Open the clear plastic door to the drive.

- Identify the type of tape drive as follows. If the capstan hub is *below* the take up reel and the switch panel has a DENSITY rocker switch, the drive is type MTB. If the capstan is below the take up reel but there is no DENSITY switch, the drive is type MTA. If the capstan is on the *right* (alongside) of the take up reel, the drive is a "stream" type MTC.

- Mount and thread the tape according to the diagram on the drive door (MTB or MTA) or above the capstand (MTC). Rotate the take up reel 4 or 5 times to get a good purchase on the tape.

- On the central panel of the drive, there are several rocker switches. The POWER switch should be ON. Press the BOT/UNLOAD switch to BOT (or on an MTA drive, press LOAD/UNLOAD to LOAD). The tape will hesitate, move forward, and stop.

- Press the ON LINE/OFFLINE switch to ON LINE.

- Note the number on the drive thumbwheel (0, 1, 2, or 3) if there is one. An MTC drive has no thumbwheel; assume the drive is number 0. With the number in mind, return to your terminal.

Having physically mounted the tape, you can dump files to it. You will use the real tape drive device name for this. Type

```
) DIR/I ↵
) DUMP/L=@LPT   { @MTBn:0 ↵ or @MT0:0 ↵ or @MTAn:0 ↵ }
```

The @ specifies the peripheral directory, home of all device entries; e.g., @LPT. n is the number of the tape drive that you noted. (If you get a "file does not exist" message, try a different drive type.)

The DUMP command copies (dumps) disk files to a destination file. If you type DUMP from your user directory and omit file (path) names, DUMP dumps *all* files from the user directory and inferior directories. In this case, the dump is to tape file 0, (@MTBn:0 or @MTAn:0) but you could have specified a disk file. Dump files can, and usually do, contain more than one disk file. The /L=@LPT switch directs a listing of all DUMPed files to the line printer.

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

```
) REWIND  { @MTBn ↵ or @MTC0 ↵ or @MTAn ↵ }
)
```

Return to the tape drive, press the RESET switch, then the UNLOAD switch to unload the tape.

After Dumping to Tape

You have dumped all your files to tape. Get the tape from the operator or drive and get the listing from the line printer. Write the *file number* on the listing, and clip the listing between the tape cover and tape reel.

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, you'll lose the original tape file and all subsequent tape files. For your next dump on this tape, dump to file 1, and then 2, and so on; if the tape is long enough, you can go to number 99.

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 dump. For example, *tomorrow* you might type

```
) DIR/I ↵
) dump/I=@LPT/after/tlm=10-JAN-81 { linkname:1↵ or @MTBn:1↵ or @MTC0:1↵ or @MTAn:1↵ }
```

to dump all files with time last modified (after/tlm) since 10 January 1981. The after/tlm switch indicates the files modified after the specified time. FILESTATUS and MOVE also offer date switches. See DUMP and FILESTATUS in Chapter 3 for more details about date switches.

If you ever need to restore your dumped files to disk, you'll use the counterpart of DUMP, which is LOAD. LOAD is further described in Chapter 3.

Having read this section, you may decide that you never need to dump files to tape. This is fine. But if you need to do it, you know how -- whether or not an operator is on duty.

Getting HELP

AOS has an extensive HELP facility, which you can use whenever you're unsure about a command or its format, its switches, or its arguments. We didn't describe HELP earlier because you need a little background to understand its messages.

Type

```
) HELP ↵
```

and you'll see a list of topics on your screen, like this:

```
*1_SWITCH          *2_SWITCH          *AFTER_SWITCH      *CLI_INPUT         *COMMANDS
*CONDITIONALS      *CONTRL_CHARS      *CURSOR_CONTROL    *DEBUG             *DISPLAY
*ENVIRONMENT       *EXCEPTIONS        *EXEC              *FED               *FILENAMES
*GENERIC_FILES     *I_SWITCH          *LINK              *LINKS             *L_SWITCH
*MACROS            *M_SWITCH          *NEWLINE           *PATHNAMES         *PED
*PSEUDOMACROS      *P_SWITCH          *QUEUES            *Q_SWITCH          *REPORT
*SED               *SWITCHES          *TEMPLATES         *TOPIC             *VSGEN

FOR MORE HELP ABOUT ANY ITEM ABOVE, TYPE 'HELP *ITEM'
```

Your own TOPICs will vary according to other software on your system; for example, you may see FORTRAN topics. The HELP topics are pretty explicit, so let's look at the COMMANDS:

```
) Hel *COMMANDS ⤸
```

```
CLI COMMANDS ARE:

ACL             ASSIGN          BIAS            BLOCK           BYE
CHAIN           CHARACTERISTICS CHECKTERMS      CLASS1          CLASS2
CONNECT         CONTROL         COPY            CPUID           CREATE
CURRENT         DATAFILE        DATE            DEASSIGN        DEBUG
DEFACL          DELETE          DIRECTORY       DISCONNECT      DISMOUNT
DUMP            EXECUTE         FILESTATUS      HELP            HOST
INITIALIZE      LEVEL           LISTFILE        LOAD            LOGEVENT
LOGFILE         MESSAGE         MOUNT           MOVE            PATHNAME
PAUSE           PERFORMANCE     PERMANENCE      POP             PREFIX
PREVIOUS        PRIORITY        PROCESS         PROMPT          PRTYPE
PUSH            QBATCH          QCANCEL         QDISPLAY        QFTA
QHOLD           QPLOT           QPRINT          QSUBMIT         QUNHOLD
RELEASE         RENAME          REVISION        REWIND          RUNTIME
SCREENEDIT      SEARCHLIST      SEND            SPACE           SQUEEZE
STRING          SUPERPROCESS    SUPERUSER       SYSID           SYSLOG
TERMINATE       TIME            TREE            TRACE           TYPE
UNBLOCK         VAR0            VAR1            VAR2            VAR3
VAR4            VAR5            VAR6            VAR7            VAR8
VAR9            WHO             WRITE           XEQ

FOR MORE HELP ABOUT ANY ITEM ABOVE, TYPE 'HELP ITEM'
```

So let's ask about ACL. Type

```
) Hel  ACL ⤸

ACL                - REQUIRES ARGUMENT(S)
 SWITCHES:          /1=   /2=   /L(=)   /Q...

FOR  MORE  HELP  TYPE
HELP/V  ACL
)
```

To find out more, let's type

```
) Hel/V  ACL ⤸

ACL \ SET OR DISPLAY THE
ACCESS CONTROL LIST...
      .....

FORMAT:     ACL  TEMPLATE  <USERNAME,..

COMMAND SWITCHES:   /1=   /2=   /L(=)   /Q
      /V  DISPLAY PATHNAME
      /K  DELETE ACL
      /D  DEFAULT ACL
)
```

As you can see, HELP messages can be really fabulous when you know just a little about the system.

Winding up the Session

This session with the CLI is nearly over. Before you log off, you might want to print the log file. First, find the log file, note its pathname, and type an identifying closing line. Then print the log file and close it.

```
) DIR ⤸
:UDD:JACK
) LOGFILE/V ⤸
:UDD:JACK:LOG.FILE
) WRITE  End of the CLI Session! ⤸
End of the CLI Session!
) QPRI  :UDD:JACK:LOG.FILE ⤸
QUEUED,  SEQ =n, QPRI=n
) LOGFILE/K ⤸
)
```

LOG.FILE is pretty large by now (check it with F/AS) and you might want to delete it. If you don't delete it, you can append to it during any CLI session by typing LOGF LOG.FILE⤸; or you can specify another log file name instead of LOG.FILE. Each log file remains in the directory where you started it.

When examining the log file, you'll notice that certain parts of the dialog are not included: for example output from TYPE commands and HELP/V ACL. The log file is still a useful record of your session with the CLI.

Logging Off

To terminate a CLI session and log off the system, type

```
) BYE ⤶
```

```
AOS  CLI  TERMINATING  date  time

PROCESS  n  TERMINATED
CONNECT  TIME  hours:minutes:seconds
USER  'JACK'  LOGGED OFF @CONn  date time

system \ TYPE NEW-LINE TO BEGIN LOGGING ON
```

BYE terminates your user process and frees the terminal for anyone (with a valid username and password) to log on. If the terminal will be unused for a long time (like overnight or over the weekend), turn it off by pressing the ON switch near the lower right of the screen.

To log on again, simply type ⤶ (turning the terminal on first, if needed), then username ⤶ and password ⤶, as described at the beginning of the chapter.

Changing Your Password

If you ever want to change your password, you can do it when you log on. Log on as usual, with username and password; but, after you type the password, press ERASE PAGE (or CTRL-L) instead of NEW LINE. The system will then ask for the new password, and you'll type it in. The password must be 3 through 15 characters long, and can include all valid filename characters: A-Z, 0-9, ?, $, _ (underscore), or . (period).

For example, take the following dialog, in which a user whose username and password are JOAN changes her password to JKN, then logs off and logs on again with her new password:

```
.... TYPE NEW-LINE TO BEGIN LOGGING ON ...

⤶
....EXEC  n  date .....
USERNAME:  JOAN ⤶                    Username ⤶ .
PASSWORD:  JOAN (ERASE PAGE)         Password and
                                     ERASE PAGE
                                     key.
ENTER YOUR NEW PASSWORD:  JKN⤶       New password⤶.

--NEW PASSWORD IN EFFECT --          System
                                     confirms.
...LAST PREVIOUS LOGON...            System logs user
                                     on.

) BYE ⤶                              User logs off.

... CLI TERMINATING ....
```

```
.... TYPE NEW-LINE TO BEGIN LOGGING ON ...

⤶
....EXEC n  date ....                Logs on again to
                                     check new
                                     password.

USERNAME:  JOAN ⤶                    Username.
PASSWORD:  JKN⤶                      New password.

...LAST PREVIOUS LOGON...            New password
                                     works.
)
```

The new password remains in effect until you change it again. If you change your password, choose one that you can remember easily -- because, if you forget it, you won't be able to log on. (If you ever *do* forget your password and/or username, consult the system operator.)

If privacy and security are very important to you, don't choose obvious names or initials for your password. Good choices include a mixture of alphabetic and numeric characters, and/or special characters; for example

```
MY_9_PASSWORD
```

Summary and Congratulations

You have finished the entire CLI chapter. You've logged on; started a log file; created and deleted files; created a directory; moved files into it; tried screen editing, learned something about filenames, pathnames, and templates; and printed a file. You've used the CTRL control characters; played with searchlists and ACLs; executed a text editor program; perhaps used MOUNT and dumped your files to magnetic tape; gotten help; closed and printed the log file; and logged off. Lastly, you learned how to change your password.

Congratulations. This session included most of the concepts and commands you'll use in your day-to-day interaction with the CLI, and it provides a sound background for all CLI commands and other software in the system.

You can find details on certain CLI commands in Chapter 3; these are also covered in the *Command Line Interpreter (CLI) User's Manual*, 093-000122.

Session Summary

Figure 2-4 is a concise summary of your session, adapted from a real log file.

Logging On

Turn terminal on if needed.

```
system \ TYPE NEW-LINE TO BEGIN LOGGING ON
 )
AOS n  EXEC n  9-JAN-81  9:32:49  @CON11
USERNAME:  JACK )
PASSWORD:  JACK )                    (Password
                                     doesn't echo.)

AOS  CLI  REV n  9-JAN-81  9:33:00

) LOGFILE LOG.FILE )
```

Directories and Files

```
) DIR )
:UDD:JACK
) CREATE  MYFILE )
) Files  MYFILE )

DIRECTORY :UDD:JACK

MYFILE

) FILES/AS  MYFILE )

DIRECTORY :UDD:JACK

  MYFILE       TXT   9-JAN-81   9:46:50    0
) DELETE/V  MYFILE )
DELETED MYFILE
) FILES/AS  MYFILE )

) CREATE/I MYFILE )
)) Greetings from MYFILE. )
)) ) )
) FILES/AS  MYFILE )

DIRECTORY :UDD:JACK

  MYFILE       TXT   9-JAN-81   9:52:50    24

) TYPE  MYFILE )
Greetings from MYFILE.
) CRE/I  MACRO.CLI )
)) WRITE□MYFILE□contains□;TYPE  MYFILE )
)) ) )

) MACRO )
MYFILE contains
  Greetings from MYFILE.
) FIELS/AS )
ERROR: NOT A COMMAND OR MACRO, FIELS
FIELS/AS

) F/AS )
```

Figure 2-4. Summary of Dialog in the CLI Session (continues)

```
DIRECTORY :UDD:JACK

  LOG.FILE     TXT   9-JAN-81   9:51:06   482
  MACRO.CLI    TXT   9-JAN-81   9:54:22    35
  MYFILE       TXT   9-JAN-81   9:52:50    24
```

A New Directory

```
) CRE/DIR  LEARNING )
) F/AS )
  LOG.FILE     TXT   9-JAN-81   9:51:06   647
  MACRO.CLI    TXT   9-JAN-81   9:54:22    35
  MYFILE       TXT   9-JAN-81   9:52:50    24
  LEARNING     DIR   9-JAN-81   9:56:44     0
) DIR  LEARNING )
) DIR )
:UDD:JACK:LEARNING
) DIR ↑ )
) DIR )
:UDD:JACK
) MOVE/V  LEARNING macro.cli myfile )
  MACRO.CLI
  MYFILE
```

Screen Editing

```
CTRL-A screen edit sequence redisplayed command line.
You typed ):
                )
WARNING: FILE... EXISTS: LEARNING:macro.cli
WARNING: FILE... EXISTS: LEARNING:myfile
CTRL-A              redisplayed command line;
HOME key            moved cursor to start of line;
DELETE/V            overwrote MOVE/V;
space bar           wiped out EARNING;
CTRL-F              moved cursor
CTRL-F              to end
CTRL-F              of line.
)                   entered edited command line.

  DELETED macro.cli
  DELETED myfile

CTRL-A        redisplayed command line;
HOME key      moved cursor to beginning of line;
CR key        deleted command line.
```

Filenames and Pathnames

```
) TY  MYFILE )
WARNING: FILE DOES NOT EXIST..MYFILE
) TY  LEARNING:MYFILE )
Greetings from MYFILE.

) DIR  LEARNING )
) TYPE  MYFILE )
Greetings from MYFILE.
) TY  LOG.FILE )
WARNING: FILE DOES NOT EXIST... LOG.FILE
) TY ↑LOG.FILE )

(text of LOG.FILE)
```

Figure 2-4. Summary of Dialog in the CLI Session (continued)

Keeping Track of Filenames

```
) F/AS/S ↓

DIRECTORY :UDD:JACK:LEARNING

 MACRO.CLI   TXT   9-JAN-81   9:54:22   35
 MYFILE      TXT   9-JAN-81   9:52:50   24

) FILES ******* ↓

DIRECTORY :UDD:JACK:LEARNING

 MYFILE

) FILE - ↓

DIRECTORY :UDD:JACK:LEARNING

 MYFILE

) FILES -.CLI ↓

DIRECTORY :UDD:JACK:LEARNING

 MACRO.CLI

) FILES M+ ↓

DIRECTORY :UDD:JACK:LEARNING

 MACRO.CLI          MYFILE

) FILES +\+.CLI ↓

DIRECTORY :UDD:JACK:LEARNING

 MYFILE

) FILES :UDD:JACK:# ↓

DIRECTORY :UDD

 JACK

DIRECTORY :UDD:JACK

 LOG.FILE           LEARNING

DIRECTORY :UDD:JACK:LEARNING

 MACRO.CLI          MYFILE

) F/AS :UDD:JACK:#:FOO ↓

DIRECTORY  :UDD:JACK

DIRECTORY  UDD:JACK:LEARNING
```

Figure 2-4. Summary of Dialog in the CLI Session (continued)

Printing Files

```
) QPR MACRO.CLI ↓
QUEUED, SEQ=29, QPRI=127
```

Check out the line printer, getting the operator to give you your file, using a form feed macro, or working the controls (ON/OFF LINE, TOP OF FORM *an even number of times*, ON/OFF LINE) if needed.

```
) FILES/AS/S/L=@LPT ↓
```

Utilities and Searchlists

```
) SEARCH ↓
:UTIL, :LINKS, :MACROS
) XEQ SED FOO ↓
Do you want FOO to be created?  NO ↓
Start over?  NO↓

) SEARCH :udd:jack:learning ↓
) sea ↓
:UDD:JACK:LEARNING

) XEQ SED FOO ↓
ERROR: FILE DOES NOT EXIST
XEQ,SED,FOO

) sea :UTIL, :UDD:JACK:LEARNING ↓
) sea
:UTIL,:UDD:JACK:LEARNING
) XEQ SED FOO ↓
Do you want FOO to be created?  NO ↓
Start over?  NO↓
```

System Control Sequences

```
) TYP :UTIL:PARU.SR ↓
(text of PARU.SR)

        .
        CTRL-S
        CTRL-Q
          .
          .
        CTRL-S
        CTRL-C  CTRL-A
        CTRL-Q

ERROR: CONSOLE INTERRUPT

) XEQ SED ↓
Name of file to edit:  CTRL-C  CTRL-B [C]B

*ABORT*
CONSOLE INTERRUPT
ERROR: FROM PROGRAM
XEQ,SED
```

Figure 2-4. Summary of Dialog in the CLI Session (continued)

File Access Control

```
) DIR :⤸
) F/AS⤸
WARNING: FILE ACCESS DENIED, FILE =
) DIR :UDD⤸
) F/AS⤸
WARNING: FILE ACCESS DENIED, FILE =
) DIR/I LEARNING⤸
) DIR⤸
:UDD:JACK:LEARNING

) ACL/V MYFILE⤸
MYFILE                    JACK,OWARE
```

Making Files Permanent

```
) CREA/DIR foo⤸
) dir foo⤸
) create FOOFIL⤸
) dir ↑⤸
) f/as foo⤸

DIRECTORY :UDD:JACK:LEARNING

 FOO                      DIR  9-JAN-81  10:12:48
                          4608

) del/v foo⤸
DELETED foo
) cre/dir foo⤸
) perm foo on⤸
) perm/v foo⤸
foo                ON
) del/v foo

WARNING:CANNOT DELETE PERMANENT FILE
) perm foo off⤸
) del/v foo⤸
DELETED foo
```

Backup for Your Files

```
) MOUNT  TAPE  Could you mount a tape?⤸
```

Unless response was *REQUEST REFUSED*, dump your files (mounting the tape first if there was no response).

```
) DIR/I⤸
) DUMP/L=@LPT  TAPE:0⤸      (If prompt returned
                             from MOUNT
or                           command)

) DUMP/L=@LPT @MTB0:0⤸      or @MTC0:0 or
                             @MTA0:0 (if
                             you mounted the
                             tape yourself)
```

(Files are DUMPed to tape file 0 .)

Figure 2-4. Summary of Dialog in the CLI Session (continued)

```
) DISMOUNT  TAPE⤸            (If prompt returned
                             from
                             MOUNT
or                           command)

) REWIND  @MTB0⤸             or @MTC0 or
                             @MTA0 (if you
                             mounted the
                             tape yourself)
```

Getting HELP

```
) HELP⤸

TOPICS ARE:

*...   *...   *...   *...   * COMMANDS
*...   *...   *...   *...   *...
*...   *...   *...   *...   *...

FOR MORE HELP ABOUT ANY ITEM ABOVE,
      TYPE 'HELP *ITEM'

) hel *COMMANDS⤸

CLI COMMANDS ARE:

ACL   ...     ...     ...     ...
...   ...     ...     ...     ...
...   ...     ...     ...     ...
...   ...     ...     ...     ...

FOR MORE HELP ABOUT ANY ITEM ABOVE,
      TYPE 'HELP ITEM'

) Hel ACL⤸

ACL              - REQUIRES ARGUMENT(S)
 SWITCHES: /1= /2= /L(=) /Q ...

FOR MORE HELP TYPE
HELP/V ACL

) Hel/v ACL⤸

ACL     SET OR DISPLAY THE ACCESS ...
        ...

FORMAT:  ACL  TEMPLATE ...

COMMAND  SWITCHES:  /1= &/2= ...
        ...
        /K  DELETE ACL.
        /D  DEFAULT ACL.
```

Figure 2-4. Summary of Dialog in the CLI Session (continued)

Winding Up the Session

```
) DIR ↵
:UDD:JACK
) LOGFILE/V ↵
:UDD:JACK:LOG.FILE
) WRITE  End of the CLI Session! ↵
End of the CLI Session!
) QPRI  :UDD:JACK:LOGFILE ↵
QUEUED,  SEQ =n
) LOGFILE/K ↵

) BYE ↵

AOS  CLI  TERMINATING  9-JAN-81  10:43:07

PROCESS  9  TERMINATED
CONNECT  TIME  1:03:06
USER 'JACK' LOGGED OFF  @CON11 9-JAN-81 ...


system \ TYPE NEW-LINE TO BEGIN LOGGING ON
```

Optionally, experiment with changing password.

Figure 2-4. Summary of Dialog in the CLI Session (concluded)

What Next?

If you want a command-by-command summary of the CLI commands you've used (and a few you haven't), read Chapter 3.

To start text editing -- for its own sake or to write FORTRAN, COBOL, or assembly language programs -- you need to learn a text editor. Proceed to a text editor chapter: Chapter 4 for SED (the easy, screen-oriented text editor) or Chapter 5 for SPEED (the powerful, character-oriented editor).

To start programming in AOS BASIC, you don't really need to know a text editor (although you may find one easier to use than the BASIC editor). You can go directly to Chapter 8.

End of Chapter

Chapter 3
CLI Features and Commands

This chapter reviews the screen-editing and system control characters and the CLI commands that were described in Chapter 2. Chapter 3 is primarily a reference chapter; the most salient topics were covered in Chapter 2. Not all features of the commands are described; use HELP/V ⤵ for more information.

Commands not introduced in Chapter 2 are DATE, LISTFILE, RENAME, SPACE, and TIME.

Screenediting and System Control Characters

Sequences that you begin by pressing the CTRL key fall into two categories: screen editing and system control. Table 3-1 describes the screen-editing sequences; Table 3-2 describes the system control sequences.

Table 3-1. Screen-Editing CTRL Sequences

CTRL-A	Redisplays the last command typed.
CTRL-F	Moves cursor forward to next word.
CTRL-B	Moves cursor backward to preceeding word.
→ key or CTRL-X	Moves forward one character.
← key or CTRL-Y	Moves backward one character.
CR or ERASE EOL key	Deletes all characters to the right of the cursor. CR executes the command.
CTRL-E	Inserts new text or terminates an insert.
DEL key	Deletes the preceding character.

Table 3-2. System CTRL Sequences

CTRL-L	Clears the screen and tells CLI to execute command (if any).
CTRL-S	Postpones display. To display all suspended information, enter CTRL-Q.
CTRL-Q	Resumes display suspended by CTRL-S.
CTRL-O	Cancels 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 (in CLI, SED, and SPEED text editors only).
CTRL-C CTRL-B	Aborts the current program, returns to its parent process. If entered from the CLI, CTRL-C CTRL-B logs you off the system.

CLI Commands

As you saw in Chapter 2, the CLI is a powerful utility. It provides over 100 commands and a macro facility so you can create still more. You can do a lot of work using just a few of these commands.

The remainder of this chapter describes the most important features of the following CLI commands:

ACL	LOGFILE
BYE	LOAD
COPY	MOVE
CREATE	PERMANENCE
DATE and [!DATE]	QPRINT
DELETE	RENAME
DIRECTORY	SEARCHLIST
DUMP	SPACE
FILESTATUS	TIME
HELP	TYPE
LISTFILE	WRITE
	XEQ

Most of these commands have features that are not described here. For a complete description of every command, see the *Command Line Interpreter User's Manual*, 093-000122.

Abbreviations

When you abbreviate a command, you can use the shortest string of letters that uniquely identifies it. Don't be afraid to experiment with short abbreviations; the worst that can happen is

ERROR: COMMAND ABBREVATION NOT UNIQUE

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

Multiple Commands and Long Command Lines

To stack more than one CLI command on a line, separate the commands with a semicolon; for example:

```
) F/AS  MYFILE; TY  MYFILE; WRITE MYFILE↓
```

To continue a command onto another line, type an ampersand and ↓; the CLI will then display an ampersand-prompt on the next line and you can continue typing. When you want the CLI to execute the command string, press ↓ without typing an ampersand. For example:

```
) TY&↓
&) PE&↓
&) □MYFILE↓
```

Extra spaces or tabs within CLI commands have no effect.

ACL

Display or set a file's access control list

Format

ACL pathname *[username,privileges] [username,privileges] [...]*

If you omit arguments, ACL displays the file's access control list. If you include arguments, they must be paired: username comma privileges. With arguments, ACL sets a new control list for the file. It gives users any or all of the following access privileges:

O	Owner access
W	Write access
A	Append access
R	Read access
E	Execute access

Generally, you must have owner access to change a file's ACL. You can assign all privileges to yourself for any file within your user directory.

Files that are dumped or moved usually retain their original ACLs, so, if you receive files from another person, give yourself OWARE privileges to them before trying to edit or otherwise process them.

You can use templates with the username portion of arguments.

Command Switches

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

Examples

```
) ACL  MYPROG.PR ↓
JACK, OWARE
) ACL  MYPROG.+   JACK,OWARE  +, RE ↓
) ACL/V  SUBDIR:FILEA↓

SUBDIR:FILEA            JACK,OWARE
```

The first ACL command displays the ACL for MYPROG.PR. The second command adds read and execute access for all users for MYPROG.PR. The third ACL command displays FILEA's pathname along with its ACL.

BYE
Log off the system

Format

BYE

BYE -- when you type it from the CLI -- terminates the user process that the system created when you logged on; this logs you off the system.

Issued from a utility like SED, BYE returns you to the CLI.

Command Switches

none

Examples

) BYE ⤸

..... *CLI TERMINATING* ...
...
USER 'JACK' LOGGED OFF ...
.....

COPY
Make a copy of one or more files

Format

COPY destination-pathname source-pathname */.../*

COPY copies the contents of the file(s) named in source-pathname to the file named in destination-pathname. If you specify two or more source-pathnames, they will be copied in the order given into destination-pathname.

If destination-pathname is an existing disk file, you must use a command switch.

You cannot use templates in arguments to the COPY command.

Command Switches

/A Append source-pathname to the end of existing file destination-pathname.

/D Delete the destination-pathname, if it exists, then copy source-pathname(s) to a new destination-pathname.

/V Verify files copied; useful when you copy many files to one.

Examples

) COPY MYFILE1 MYFILE ⤸

This command creates MYFILE1, then copies MYFILE to it.

) COPY FILE1 ↑FILE2 ⤸

This creates FILE1 in the working directory and copies FILE2 (in the superior directory) to FILE1.

) COPY/A LIB1 FILE1 FILE2 ⤸

This opens file LIB1 and appends FILE1 and FILE2 to it.

069-000018

CREATE
Create a file

Format

CREATE pathname

CREATE creates a file. The new file is an individual file unless you use the /DIRECTORY switch.

With the /I switch, you can write text into a new nondirectory file. After you type CREATE/I and press ↵, the CLI displays two right parentheses [ ))] as a prompt for input. To end each line of text, press ↵. To end the insert, type)↵ next to the CLI)) prompt.

Command Switches

/DIRECTORY	Create a directory file.
/I	Create file and open for input.

Examples

```
) CREATE/DIR  NEWDIR ↵
```

This creates a new directory file named NEWDIR in the working directory.

```
) CREATE/I  FAS.CLI ↵
)) WRITE Am executing FILES/AS/S ↵
)) FILES/AS/S↵
)) )↵
)
```

This command creates a CLI macro that does a FILES/AS/S command. To use it, simply type FAS↵.

DATE and [!DATE]
Display date and expand to date

Format

DATE

...[!DATE]...

The DATE command displays the current system date.

[!DATE] belongs to a family of CLI constructs called *pseudo-macros*. Pseudo-macros are designed to help you write CLI macros. Used within a command, they expand to a specific value. [!DATE] expands to the system date. You might use [!DATE] as shown in the second example below.

Command Switches

none

Examples

```
) DATE ↵
9-JAN-81
```

The system date is 9 January, 1981.

```
) CREA/I  TODAY.CLI↵
)) WRITE Today's files are; ↵
)) FILES/AS/S/AFTER/TLM=[!DATE]↵
)) )↵
)
```

Macro TODAY.CLI, executed by typing TODAY↵, will display the message *Today's files are* followed by an alphabetical list of filenames that were created or modified today. You could use a variation of TODAY with the DUMP command to dump all files created or modified today.

DELETE
Delete one or more files

Format

DELETE pathname *[pathname][...]*

DELETE can delete both individual and directory files.

If you delete a directory, every individual file within the directory will be deleted. The system will not, however, allow you to delete a directory that has inferior directories.

You can use filename templates with DELETE.

To protect files from deletion, use the PERMANENCE command.

Command Switches

/C Confirm. The CLI will display each filename and wait. If you type Y⤶, CLI will delete the file; if you type any characters other than Y⤶, the CLI will not delete the file.

/V Verify: display the name of each deleted file.

Examples

```
) DEL/V  FOO⤶
DELETED     FOO
) DEL/V  SUBDIR:FOO⤶
DELETED     SUBDIR:FOO
```

These DELETE commands deleted file FOO 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
```

This DELETE command directs the CLI to delete all files in the working directory whose names begin with TEST, followed by any single character except a period. With /V, you tell the CLI to verify each deletion; with /C, you ask the CLI to confirm each deletion. The CLI displays each matching name and waits. In the dialog, you delete two files and retain two others.

DIRECTORY
Display or set the working directory name

Format

DIRECTORY *[dir-pathname]*

If you omit switches and arguments, DIRECTORY displays a complete pathname to the working directory.

If you include a *dir-pathname* argument, DIRECTORY makes the specified directory the working directory. If you omit the /I switch and the desired directory is superior to the working directory, *dir-pathname* must be a full pathname from the root (:). If the desired directory is inferior to the working directory, *dir-pathname* can start with a directory within the working directory.

If you include the /I switch, DIRECTORY makes the initial (user) directory or one of its inferiors the working directory.

Command Switches

/I Set working directory to initial directory or, if you include *dir-pathname,* to *dir-pathname.*

Examples

```
) DIR ⤶
:UDD:JACK
) DIR  LEARNING⤶
) DIR ⤶
:UDD:JACK:LEARNING
) DIR/I ⤶
) DIR ⤶
UDD:JACK
```

The first DIRECTORY command shows you the working directory name, JACK. Next we make an inferior directory, LEARNING, the working directory. And finally, DIR/I makes the initial (user) directory the working directory.

DUMP
Dump one or more files

Format

DUMP dump-pathname *[source-pathname] [source-pathname] [...]*

Periodically, you should save copies of important files with the DUMP command. The best medium for storage is magnetic tape. Later, if need be, you can restore any or all of the dumped files with the LOAD command. You can also use DUMP to make copies of your files for other people.

Although the COPY command can copy files, DUMP is preferable because it

- Can dump files from a directory and inferior 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.

If you omit *source-pathnames,* the DUMP command copies *all* files (or all those selected by date switches) in the working directory *and inferior directories.* If you include *source-pathname(s)* and they do not specify a directory, the DUMP command copies only those files in the working directory.

DG tape drives have the device names @MTxn where x is B for newer models and A for others; and where n is the number dialed on the drive thumbwheel. The *tape file number* applies to the tape you are using. Typical drive names with file numbers are @MTB0:0 or @MTA1:0 .

Command Switches

/AFTER/TLM=dd-mmm-yy	DUMP all files created or modified on or after day dd in month mmm in year yy. 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 *[=@LPT]*	List pathnames dumped to the LISTFILE, or to the line printer if you say /L=@LPT. You must also include /L.
/V	Verify names dumped to the console.

Examples

```
) DUMP/V  @MTB0:0 ↓
```

.
.

This command copies all files in the working directory and inferior directories to file 0 of the mag tape mounted on tape drive 0; it verifies each pathname dumped. Files on mag tape proceed sequentially; first you dump to file 0, then file 1, and so on. The command DUMP @MTB0↓ would do the same thing, 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 the working directory* to the second file of the tape on drive @MTC0. It lists files dumped to the line printer.

```
) DIR/I ↓
) dump/after/tlm=9-JAN-81/v/l=@lpt  @mtb0 ↓
```

This command dumps all files in and below the user directory that were created or modified after 9 JANuary 1981. The files are dumped to file 0 of the tape on drive @MTB0 and their names are listed on the line printer. See DATE for a macro that you can adapt for this kind of thing.

FILESTATUS
List file names and statistics

Format

FILESTATUS *[pathname] [pathname] [...]*

FILESTATUS displays information on files in and below the the working directory. If you don't give a pathname, the command applies to files in the working directory. If you use a directory name in the pathname, the command applies to all files within that directory; it will also display a header that describes the directory.

You may use templates in arguments to the FILESTATUS command.

Command Switches

/AFTER/TLM=dd-mmm-yy	Include all files created or modified on or after day dd in month mmm in year yy. 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.
/ASSORTMENT	Include an assortment of information: filename, type, date and time created, and size in bytes.
/BEFORE/TLM/=dd-mmm-yy	Include files created or modified before the given day. Has same form as /AFTER.
/L */=@LPT]*	Write filename to the list file, or to the line printer if you use /L=@LPT .
/TYPE=DIR	Include only directory files.

Examples

```
) FILES  TEMP+ ↓
```

This command lists on the terminal the name of every file that begins with TEMP in the working directory.

```
) FILES/AS/S  +.F77 ↓
        .
        .
```

Within the working directory, display an assortment of information about all files whose names end in .F77 . Sort the filenames alphabetically before displaying them.

```
) FILES/AS/TY=DIR ↓
```

Display names and other assorted information on all directories within the working directory.

```
) DIR/I ↓
) F/AS/S  # ↓
```

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

HELP
Display HELP information

Format

HELP *[item] [item] [...]*

If you type HELP without an argument, it displays a list of items that relate to DG software on your system. HELP messages try to be self-explanatory and guide you to the next HELP construct. To summarize:

- To get the names of all CLI commands, type HELP *COMMANDS↵.
- To get the name of every CLI command that begins with a letter combination, type HELP combination.
- To get all HELP information on a command, type HELP/V command↵.

If a HELP message is too big for your terminal screen, use CTRL-S and CTRL-Q to read it segment by segment.

HELP is shown in action in Chapter 2, under "Getting HELP."

Command Switches

/L *[=@LPT]*	Write the HELP information to the current list file, or the the line printer if you use /L=@LPT .
/V	Give a verbose description. Without /V, you'll see a terse description.

Examples

```
) HELP  L↵
L- POSSIBILITIES   ARE:
LEVEL
LISTFILE
LOAD
LOGEVENT
LOGFILE

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

The first HELP command displays the names of CLI commands that begin with L . The second HELP command displays a terse message on command LISTFILE.

LISTFILE
Display or set the generic @LIST filename

Format

LISTFILE *[pathname]*

The LISTFILE command relates to one of the system files called a *generic file*. A generic file is really a pointer to a filename. You can set the filename yourself, according to your needs.

The pointer to the list file is called @LIST; but the file itself does not initially exist. The LISTFILE command sets it and creates it if necessary. For example, either of the following are valid list files.

> @CONSOLE (your terminal); or
> any legal disk file pathname.

LISTFILE is most useful with programs that you write, compile, and build 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 file at will; you need not change the source, recompile, and rebuild.

User programs (and the CLI) access the file pointed to by @LIST just as any other file. (But if a program tries to open @LIST and you have forgetten to set the list file, the program may bomb with a "FILE DOES NOT EXIST" message.)

If you specify an 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.

Several of the example programs later in this book use LISTFILE.

Command Switches

none

Examples

```
) LIST ↵
@LIST
) LIST   @CONSOLE ↵
) XEQ    MYPROG ↵
          .
        (output to terminal)
          .
) LIST   MYPROG.OUTPUT.FILE ↵
) XEQ    MYPROG ↵
```

The first LISTFILE command checks the list file; it is @LIST, which means it isn't set. The next LISTFILE command makes the list file @CONSOLE; then we execute MYPROG and output goes to the terminal. The output looks ok, so we make the list file a disk file named MYPROG.OUTPUT.FILE and output goes to MYPROG.OUTPUT.FILE.

LOGFILE

Display the log filename or start recording in a log file

Format

LOGFILE *[pathname]*

If you omit *pathname*, LOGFILE displays the current log file name, or a null if there is no current log file.

If you include *pathname*, the CLI opens *pathname* and writes any dialog from the terminal to it. If *pathname* doesn't exist, the CLI creates it; if it does exist, the CLI appends to it. If you include *pathname* and there is an active log file, the CLI closes the old log file and opens the one specified in *pathname*.

When you have started a log file, it records dialog until you

- type LOG/K ↵;
- start a different log file; or
- log off the system.

You can continue output to any log file by specifying its name in a LOGFILE command.

All CLI dialog, except output from TYPE statements and HELP/V commands, is written to the current log file. Terminal I/O with other programs, such as text editors or compilers, is *not* recorded in the log file.

Command Switches

/K Set the log file to null; i.e., stop recording in it and close it.

/V Display the current log file's pathname.

Examples

```
) LOGF/V ↵

) LOGF   MY.LOG↵
          .
  (Dialog with the CLI.)
          .
) LOGF/K ↵
) QPRI   MY.LOG ↵
```

The first command checks the current log file name; there is none. The next command opens and starts recording dialog in file MY.LOG in the working directory. CLI dialog follows; then log file MY.LOG is closed and printed.

LOAD
Load one or more files into the working directory

Format

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

LOAD complements the DUMP command. It copies previously dumped files from **dump-pathname** into the working directory. These can be files you dumped earlier for safekeeping or files that someone else dumped and gave you.

If you omit *source-pathname(s)*, and the /N switch, the CLI tries to load all files in **dump-pathname**. Any directory structure within *source-pathname* is retained as it was during the DUMP.

If an existing file has the same pathname as a file in **dump-pathname**, the CLI will signal an error and will 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.

Command Switches

/L */=@LPT]*	List on the list file the names of all files loaded, or list to the line printer if you say /L=@LPT.
/N	Display the pathnames in **dump-pathname** without loading them. This identifies the files in the dumpfile for you.
/RECENT	If a filename in **dump-filename** has a more recent creation date than the same filename in the working directory, delete the existing file and load the newer one.
/V	Display the names of all files loaded (ignored if you also specify /L).

Examples

```
) LOAD/V  @MTB0:0 ↵
```

This command tries to load into the working directory all files and directories in the first file on the mag tape mounted on drive 0. LOAD/V @MTB0↵ would have the same effect, defaulting to file 0.

```
) LOAD/V  @MTA0:1  -.F77  -.SR ↵
```

This command attempts to load into the working directory every FORTRAN 77 and assembly language source file from tape file 1 on @MTA0. It pertains only to files within the working directory at the time of the DUMP command. To search any inferior directories that were dumped, you'd type

```
) LOAD/V  @MTA0:1  #:-.F77  #:-.SR ↵
```

MOVE

Move one or more files

Format

MOVE destination-pathname source-pathname

[source-pathname] [...]

MOVE moves a copy of one or more files to the directory specified in destination-pathname. The source-pathname(s) must either be in the working directory or must be inferior to the working directory. You must have append access to the directory specified in the destination-pathname.

MOVE can copy both individual files and directories. If you omit source-pathnames, the system copies the entire directory structure, from the working directory down, to the directory named in destination-pathname.

You can use templates with source-pathnames.

Command Switches

/AFTER/TLM=dd-mmm-yy	Move all files created or modified on or after day dd in month mmm in year yy. 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	Move files created or modified before the given day. Has same form as AFTER.
/L *[=@LPT]*	List pathnames moved to the list file, or to the line printer if you say /L=@LPT .
/RECENT	If a source-pathname is more recent than a file with the same name in destination-pathname delete the older file and move in a copy of the newer one.
/V	Verify that the files have been moved by displaying their names on the terminal. (This switch is ignored if you also use /L.)

Examples

```
) MOVE/V   FORT   +.F77 ↓
BOOMER.F77
BIORHYTHM.F77
ERROR: FILE ... EXISTS: FORT:MPROG.F77
) MOVE/V/R  FORT   +.F77 ↓
MPROG.F77
)
```

In this example, we try to move every file ending with .F77 in the working directory to inferior directory FORT. The CLI moves and confirms two files, but reports an error on MPROG.F77 because MPROG.F77 already exists in FORT. Then we add the /R (RECENT) switch, which *does* move MPROG.F77 -- because the MPROG.F77 in the working directory is more recent (newer) than the MPROG.F77 in FORT.

PERMANENCE

Display or set the permanence attribute for one or more files

Format

PERMANENCE pathname { *[ON]* | *[OFF]* }

The permanence attribute, if on, prevents an individual file or directory from being deleted by any user.

If you omit *ON* and *OFF*, PERMANENCE displays the permanence setting. If you include *ON* or *OFF*, PERMANENCE sets the attribute to ON or OFF.

You can use templates with pathname.

Because text editors and other utilities internally delete files and recreate them, PERMANENCE can interfere with everyday program development. So you may want to set PERMANENCE ON only for directories and selected files. Note that even a permanent file will be deleted if its parent directory is nonpermanent and you delete the directory.

Command Switches

/V Display the file's name along with its permanence attribute.

Examples

```
) CREA/DIR  ZDIR ↵
) PERMANENCE/V  ZDIR ↵
ZDIR                    OFF
) DEL/V  ZDIR ↵
DELETED   ZDIR
) CREA/DIR  ZDIR ↵
) PERM  ZDIR  ON ↵
) PERM/V  ZDIR ↵
ZDIR                    ON
) DEL/V  ZDIR ↵
WARNING:CANNOT DELETE PERMANENT FILE
)
```

In this sequence, we create a directory, check its permanence attibute (OFF) and delete it. We then recreate it, set PERMANENCE ON, and try to delete it again. The CLI refuses to delete it.

QPRINT

Copy one or more files to the line printer

Format

QPRINT pathname *[pathname]*

QPRINT copies to the line printer the contents of files you specify in pathname .

The output on the line printer includes a header page containing your username, the file(s)' pathnames, and the time and date.

Command Switches

/COPIES=n print n copies of the file (from 1 through 24).

Examples

```
) QPRI  FILE1 ↵
QUEUED,   SEQ=92,  QPRI=127
```

This command prints the contents of FILE1 on the line printer.

RENAME
Rename a file

Format

RENAME old-pathname new-filename

The RENAME command renames a file that you specify in old-pathname, giving it the name new-filename.

RENAME is handy when you want to save an initial version of a file that will change. Each DG compiler produces a filename.OB version of a source filename, deleting the old filename.OB (if any) first. The Link utility produces a filename.PR version from the .OB, deleting the old filename.PR (if any) first.

So, assume that you have a working version of a FORTRAN program named MPROG and you plan extensive changes to MPROG. You want to keep the working files of MPROG intact. You might say:

```
) COPY   MPROG.OLD.F77    MPROG.F77 ⤸
) RENAME  MPROG.OB   MPROG.OLD.OB ⤸
) RENAME  MPROG.PR   MPROG.OLD.PR ⤸
```

Then you could change MPROG.F77, recompile, and relink it without deleting the old .OB and .PR files.

Command Switches

none

Examples

```
) RENAME   JONES   SMITH ⤸
```

This command changes the name of file JONES in your working directory to SMITH.

```
) RENAME   DIR2:FRITZ    FRITZ.OLD ⤸
```

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

SEARCHLIST
Display or set your search list

Format

SEARCHLIST *[directory-pathname] [,directory-pathname] [...]*

If you omit arguments, SEARCHLIST displays your search list.

If you include arguments, the system will set a new search list. Each *directory-pathname* argument must be a full pathname, starting at the root (:) and ending in a directory filename. If you specify more than one pathname, separate the pathnames with a comma.

The search list applies to most CLI commands that involve files (but not to DELETE, DUMP, FILESTATUS, or MOVE) and to CLI macros. When you type one of these, the CLI first searches the working directory. Then it scans the search list from top to bottom, searching the directories specified there. The search ends when the CLI finds the first file with the name you have supplied, or when it can't find any file by that name.

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

=	equals sign means working directory;
@	commercial "at" sign specifies peripherals directory; e.g., @LPT, @MTB0, @CONSOLE.
↑	uparrow means the superior directory;
:	colon is part of a directory pathname.

Command Switches

none

Examples

```
) MYMACRO ⤸
ERROR: NOT A COMMAND OR MACRO, MYMACRO

) SEARCH ⤸
:UTIL, :
) SEA   [!SEARCH], :UDD:JACK:MACROS, : ⤸
) SEA ⤸
SEA   :UTIL, :, :UDD:JACK:MACROS
) MYMACRO ⤸
```

First, we try to execute a macro that we know exists; but the CLI can't find it. So we check the search list and find that directory MACROS, where MYMACRO resides, is not there. So we set the search list to include MACROS (using pseudo-macro !SEARCHLIST, which specifies the current searchlist). Then we try MYMACRO again and it executes.

SPACE

Display the amount of disk space used and remaining

Format

SPACE *[:UDD:user-directory]*

As a user, you have a fixed amount of disk storage space for all your directories and files. The amount was established by the person who created your user profile.

The SPACE command tells you how much space you have (MAX), how much is occupied by files (CUR) and how much remains free (REM).

SPACE works only with *control point directories.* All initial user directories and the root (:) are control point directories, so SPACE works with any of these. You can omit arguments if the working directory is your user directory.

Each disk block can store 512 bytes (512 characters).

Command Switches

none

Examples

```
) DIR  LEARNING ⤸
) SPACE ⤸
WARNING: ...NOT A CONTROL POINT DIRECTORY...
) DIR/I ⤸
) SPACE ⤸
MAX 15000,   CUR    39, REM 14961
```

In this example, we type SPACE from an inferior directory and get an error message. Then, after making the initial user directory the working directory, we try SPACE again and get results. Our username was alloted 15,000 disk blocks; 39 are currently occupied by files, and 14,961 blocks remain free.

TIME

Display the system time

Format

TIME

The TIME command displays the current hour (of 24), minute, and second. 12 midnight is 00:00 hours, 12 noon is 12:00 hours, and 6:00 pm is 18:00 hours.

Command Switches

none

Examples

```
) TIME ⤸
14:17:06
```

The current time is 17 minutes and 6 seconds after 2 P.M.

TYPE
Copy one or more files to the screen

Format

TYPE pathname *[pathname] [...]*

TYPE copies the file named in pathname to the screen.

TYPE works properly only for text (ASCII) files. If you use TYPE on a file whose name ends in .PR, .OB, .ST, or .LB, you'll get strange results.

As mentioned earlier, you can use CTRL-S and CTRL-Q to suspend and resume display of text on your screen. To discard output -- a time saver on printing terminals -- use CTRL-D.

Command Switches

none

Examples

```
) TYPE  FILE2 ⤸
          .
```

text of FILE2

```
          .
) TY  :UTIL:PARU+
          .
```

text of parameter file

```
          .
```

These commands type FILE2 and the user parameter file on the screen.

WRITE
Display arguments

Format

WRITE *[argument] [argument] [...]*

WRITE displays arguments on the screen. The argument is either a text string or one of the CLI's pseudo-macros (defined under DATE). WRITE is useful for describing what is happening within macros, for documenting log files, and for testing other command line contructs not described here.

If you omit arguments, WRITE displays a blank line. Generally, avoid using CLI punctuation -- such as the semicolon -- *within* WRITE commands because the CLI will interpret them. Of course, you can and will use them *between* multiple commands on one line.

Command Switches

none

Examples

```
) CREA/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 My username is [!USERNAME] and⤸
)) write my process ID is [!PID].⤸
)) write The working directory is; [!DIR] ⤸
)) write My search list is [!SEARCHLIST].⤸
)) write Current space in :UDD:[!USERNAME] is ⤸
)) space :UDD:[!USERNAME] ⤸
)) )⤸
)
```

This example shows a macro, ?.CLI , which, when executed via ?⤸ will display information on all system users and additional information on yourself. The WHO command and bracketed pseudo-macros are further described in the HELP messages and in the CLI User's manual. (There's a lot of text in this macro -- so, if you want to try it, you probably should use a text editor to type it in. The SED editor is described in the next chapter.)

XEQ
Execute another program

Format

XEQ pathname *[argument]*

The XEQ command executes the program file named in pathname. A program file is a file that has been compiled/assembled and built into an executable program; its name always ends in .PR . You can omit the .PR from the pathname.

The optional argument, typically is the name of a file that you want the program to find and process. It may also be an actual argument to the program that you want to run.

Generally, you supply an argument when you execute system utility programs like text editors (SED.PR) and Link (LINK.PR).

When you write and build a program, it has the same filename you gave the source file; the Link utility replaces the appropriate original three-character extension (if any) with .PR . For example, source file MPROG.F77 or MPROG produces program file MPROG.PR . But if you gave MPROG an unconventional extension, like MPROG.FT, the compiled object module would be MPROG.FT.OB; after you ran Link on it, the program file would be MPROG.FT.PR, and you would execute it as MPROG.FT .

Command Switches

none

Examples

```
) XEQ  SED  BOOMER.F77 ↓
Do you want BOOMER.F77 to be created?  Yes↓
.
* .... ↓        (SED editor commands)
  .... ↓
* BYE ↓
)
```

In this example, you execute a text editor to write a FORTRAN 77 source program. FORTRAN 77 includes CLI macros that will compile and link BOOMER into BOOMER.PR, so you need not execute the compiler and Link directly. Finally, you execute program BOOMER:

```
) X  BOOMER ↓
```

What Next?

This was primarily a reference chapter. If you read it sequentially, and this is your first experience with the system, proceed to a text editor chapter: Chapter 4, SED (the screen-oriented editor) or Chapter 5, SPEED (the character-oriented editor).

End of Chapter

Chapter 4
Writing Text with the SED Editor

This chapter shows you how to use the SED editor, in a working session like the session with the CLI. The SED commands are summarized individually near the end of the chapter.

As with the CLI, the best way to learn SED is to use it. So, we'll describe essentials, then start using the editor.

To create and edit a file with SED, you must have append and write access (as you always have for files you create within your user directory).

The SED prompt is an asterisk (*) at the top of your screen (we assume you're using a display terminal). You type SED commands next to the * prompt. In this chapter, (ESC) in the text means that you press the ESC key. (ESC) changes command mode, but does not echo (appear) on your terminal when you press it.

Cursor and Case of Characters

The *cursor* (on display terminals) indicates your 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.

You can type SED commands in either upper- or lowercase. All compilers accept either case in text strings, but earlier FORTRANs (not FORTRAN 77) and COBOL require *statements* to be in uppercase. In the session, we use uppercase for SED commands and abbreviations; we use upper- and lowercase for the text.

If You Make a Mistake

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

A Session with SED

To illustrate SED commands, we'll create and edit a prototype source program named MYPROG.TOY . The dialog includes several deliberate errors which you should enter as shown.

Log on to the system, if you have not logged on. Ensure that the working directory is your user directory by typing

```
) DIR/I ↓
)
```

Execute SED by typing

```
) X  SED  MYPROG.TOY ↓
```

If you get an *ACCESS DENIED,X,SED* error message, omit XEQ from the command line and try again. SED announces itself and you tell it to create the file as follows:

```
SED   Rev. n  Input file -    MYPROG.TOY
Do you want MYPROG.TOY to be created?    Y ↓
*                                        I
```

By saying Y↓, you told SED to create the file. (If you had said N↓ or ↓, SED would have asked you if you wanted to *Start over?* . Usually, it's easiest to answer ↓ to the *Start over?* question and begin again from the CLI, where you can check file names with FILES/AS.)

The borderline at the top of the screen, next to the prompt, delimits the command field. The rest of the screen is free for text display. From now on we won't show the borderline.

Appending and Listing Text

To append text to MYPROG.TOY, type

```
* APPEND ↓
```

When SED is executing a command, the prompt and command remain at the top of the screen. SED always displays line numbers (unless you tell it not to), so the the text field looks like this:

```
I
```

The cursor is next to the *1*. Type in some prototype source lines:

```
        This is source line 1.↵
2       This is spurce line 2.↵
3       This is source line 3.↵
```

(ESC)

The APPEND command (shortest abbreviation is AP) allows you to append text to a file. Here, you're appending text from the terminal. To append text from a disk file, you can say APPEND FROM pathname.

The line numbers followed by tabs are for your convenience; they do not become part of the file.

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.

Type

```
* LIST  ALL ↵
```

```
1       This is source line 1.
2       This is spurce line 2.
3       This is source line 3.
```

The LIST command (shortest abbreviation is L) allows you to see a range of text. ALL (shortest abbreviation is A) is a modifier that specifies the range -- here, ALL lines in the file.

Continue:

```
* AP ↵
```

```
4       This is line 4.↵
5       This is source line 6.↵
```

(ESC)

Modifying (Editing) Text

Now let's fix the typo in line 2. Type

```
* MOD  2↵
```

```
2       This is spurce line 2.
```

```
  CTRL-F  CTRL-F  →
```

The two CTRL-Fs move the cursor to *spurce* and the → moves it to the incorrect character, *p*. Type

```
        o↵
```

```
2       This is source line 2.
```

```
3       This is source line 3.
```

The o overwrites the *p* and the ↵ (NEW LINE) told SED that you were done with the line. SED then displayed the next line, placing the cursor on *T*.

Next, let's insert source in line 4. Type

```
↵
```

```
4       This is line 4.
```

```
  CTRL-F  CTRL-F  CTRL-E  source□  CTRL-E ↵
```

```
4       This is source line 4.
```

```
5       This is source line 6.
```

Your ↵ skipped the current line (3) and displayed the next line (4) for editing. Two CTRL-Fs moved the cursor to *line*. You started a text insert with CTRL-E, typed source□ and ended the text insert with CTRL-E. Then you pressed ↵, which told SED you were done with the line. With experience, you'll be able to do this kind of thing in just a few seconds.

MODIFY is *the* SED character editing command. If you omit an argument, SED assumes the current line; if you specify a line number (or expression like +20 or -10), SED assumes the specified line. In any case, SED stays in MODIFY mode, displaying lines sequentially, until you press ESC or reach the end of the page. So press

(ESC)

As you modify, SED places the cursor at the beginning of each line. You then use cursor control characters to edit the line; and when you are done, you press ↵ to tell SED that you are done.

The cursor control/screen-editing characters work the same way for SED as for the CLI. They work in APPEND, MODIFY and INSERT commands, and while you are entering SED commands. They are

CTRL-A	Redisplays the last command or the last text appended or inserted. In MODIFY mode, CTRL-A moves the cursor to the end of the line. In APPEND or INSERT mode, it repeats the last line you typed.
CTRL-F	Moves cursor 1 word forward.
CTRL-B	Moves cursor 1 word backward.
→ key or CTRL-X	Moves forward one character.
← key or CTRL-Y	Moves backward one character.

↑ or ↓ key	Moves up or down one line. In Modify mode, maintains current column position as it moves up or down a line.
CR or ERASE EOL key	Deletes all characters to the right of the cursor. CR continues to the next line.
CTRL-E	Inserts new text or terminates an insert.
TAB key or CTRL-I	Produces a tab. Tabs are mandatory before statements in several programming languages.
CTRL-S	Suspends display; useful during a LIST ALL command for a big file.
CTRL-Q	Resumes a display suspended by CTRL-S.
DEL key	Deletes the preceding character.
HOME key or CTRL-H	Moves cursor to beginning of line.

Inserting and Deleting Text

At this point, we've appended, listed, and modified text in file MYPROG.TOY. The text looks like this:

```
1       This is source line 1.
2       This is source line 2.
3       This is source line 3.
4       This is source line 4.
5       This is source line 6.
```

Insert a line between 4 and 5 by typing

```
* IN  5↵
```

```
5       This is source line 5.↵
```

```
6
```

INSERT inserts one or more lines before the specified line; if you omit a line number, it inserts text before the current line. INSERT mode continues, with SED displaying updated line numbers, until you hit ESC. So, press

(ESC)

To insert text from a disk file instead of the terminal, you can say INSERT FROM pathname↵.

Delete line 6 by typing

```
* DEL  6↵
```

```
.       .
5       This is source line 5.
```

DELETE deletes the line (or range of lines, like 4 6) you specify. If you omit line numbers, DELETE deletes the current line. If you make a mistake with DELETE, SED lets you fix it: type

```
* UNDO↵
```

```
.       .
6       This is source line 6.
```

UNDO is useful when you accidentially delete desired text. Having undone the DELETE, delete line 6 again:

```
* D  6↵
```

Command not unique, correct the command:

```
* DE  6↵
```

```
.       .
5       This is source line 5.
```

As you can see, abbreviations that are too short do no harm; SED simply displays an error message and beeps. Just enter more of the whole command name. (CTRL-A is handy here: it redisplays the command and then you can use cursor keys, type only one or so additional letters, and type ↵ to execute the command again.)

Finding Text

Finding text strings is an important part of the editing process. To try it, type

```
* F  '1.'↵
```

String not found, correct the command:

```
* F  '1.'  ALL↵
```

```
1       This is source line 1.
```

The FIND command searches for a text string, and, if found, highlights the line within surrounding text. If you omit a range argument (e.g., 20 LA or AL), SED searches from the current line position onward.

Your first F command failed because the line position was line 5 (after the DELETE), and text string 1. doesn't exist after line 5. The second F command specified ALL lines, so SED searched all lines and found 1.

You can use either apostrophes ('string') or quotes ("string") to find a textstring.

You have now executed SED and created a file, and appended, listed, and modified text with cursor control characters. You've inserted and deleted lines, and found text. The commands involved were XEQ, APPEND, LIST, MODIFY, INSERT, DELETE, and FIND. These commands -- and BYE -- are the most common and will allow you to edit any text file.

The commands that follow simply make editing easier.

Setting POSITION

By default, the current line position is at the last line appended, inserted, or modified, or just after the end of the line deleted or found with FIND. SED displays a range of lines around the current line, and the line itself is *bright* on your screen. The POSITION command changes the current line position.

Type

```
* F  '2.'↵
```

String not found, correct the command:

```
* PO  1 ; F  '2.'↵
```

```
2       This is source line 2.
```

Here, trying to find 2., we hit the same error as before. But, instead of saying F '2.' ALL↵, we set position on line 1 and said F '2.'↵. The end result is the same; the search started before the string 2. so it found 2. . But POSITION searches are a little more limited than ALL searches because they can't scan the first line. (The semicolon simply allows you to stack SED commands, as with the CLI.)

POSITION is useful when you want to insert text, and for the commands described next.

Moving and Duplicating Text

The text in MYPROG.TOY looks like this:

```
1       This is source line 1.
2       This is source line 2.
3       This is source line 3.
4       This is source line 4.
5       This is source line 5.
```

Type

```
* MOVE  2 BEFORE  1↵
```

```
1       This is source line 2.
2       This is source line 1.
```

```
* MOV  2 BEF  1↵
```

```
1       This is source line 1.
2       This is source line 2.
.       .
```

```
* MOV  1  3 ONTO MYPROG.FOO↵
```

FILE MYPROG.FOO CREATED

```
1       This is source line 4.
2       This is source line 5.
```

```
* PO  1↵
* IN  FROM  MYPROG.FOO↵
```

```
1       This is source line 1.
2       This is source line 2.
3       This is source line 3.
4       This is source line 4.
5       This is source line 5.
```

As you saw, MOVE can move a range of text elsewhere in the file (two MOVE ... BEFORE commands). And it can move a range of text into another file, creating the other file if needed. If the other file already exists, MOVE *appends* the text to it. Having moved the text out of the file, we then set POSITION to restore it and inserted it. File MYPROG.FOO remains on disk in the working directory,

Type

```
* DUP  2  5  ONT MYPROG.FOO↵
```

DUPLICATE works the same way as MOVE, but doesn't remove the specified lines; it simply copies them to the destination, within or outside the file. File MYPROG.FOO now has seven lines of text: three were moved, and four duplicated.

The destination argument in MOVE or DUPLICATE can be BEFORE line-number, AFTER line-number, or a file pathname.

MOVE and DUPLICATE, in conjunction with APPEND or INSERT, can save time with files that have many occurrences of similar text strings. You can simply copy the text in the file as needed.

Substituting Text

Type

```
* SUBS  'new source' for  'source' ALL ↵

1       This is new source line 1.
2       This is new source line 2.
3       This is new source line 3.
4       This is new source line 4.
5       This is new source line 5.
```

SUBSTITUTE allows you to substitute one text string for another text string within a specified range. SED displays each changed line. If you omit a range argument, SED changes every matching text string from the current line onward.

Displaying Edit Status

Type

```
* DIS ↵

Edit file name -    :UDD:JACK:MYPROG.TOY

Current Page:       1
Current Line:       5

Last Page:          1
Last Line:          5

View range:         10
Display mode:       0

Typer mode:         ON
Blank mode:         OFF
Upper mode:         OFF

Line numbers being displayed
```

DISPLAY gives quite a lot of information, as you can see: file pathname, current page, current line (on page), last page, last line (on current page), and other items we won't explain here.

(The *page* will always be 1 unless you paginate the file with commands not described here. A page can have up to 1,024 lines, 15 or so line printer pages -- enough for most programs. Most DG compilers report errors by line number within the file, so -- if a source program is all on one page -- you can find erroneous lines easily via SED's line numbers.)

The DISPLAY command is most useful when you forget the file name or the number of lines there are.

SED Function Keys

Across the top of your terminal keyboard is a row of function keys. A plastic cutout called a template fits over the keys, labelling their functions. If you don't have a SED template, check the SED User's Manual; the template is part of that documentation.

These function keys can make editing a lot easier. To illustrate, we'll need to make the file a little longer. Type

```
* DUP AL ON MYPROG.FOO; DU A O MYPROG.FOO↵
.
* AP FROM MYPROG.FOO ; AP FROM MYPROG.FOO↵
.           .
.           (new text)
```

Adding two MYPROG.TOYs to MYPROG.FOO, then two MYPROG.FOOs to MYPROG.TOY, makes the file long enough.

Now, look at the row of function keys. Press the leftmost key.

SED displays text, and you should note the new current line. You'll see that pressing the leftmost key moved the position backward.

Now press the *second* function key, directly to the right of the first, and note the new current line again. Press the second key again and note the new current line.

You'll see that sequential use of the second function key moves the position forward one entire screen; and sequential use of the first key moves the position *backward* one entire screen.

These function keys are very handy when you want to scan or review text. If you didn't use them, you'd have to type sequential PO +20↵ commands, or type LIST ALL↵ and use CTRL-S and CTRL-Q. Using the function keys is easier than either of these.

Getting HELP

Like the CLI, SED has a HELP command which you can use to get details on SED commands. As with the CLI, we didn't talk about HELP earlier because you need some background to understand its messages.

Type

```
* HELP⤸

ESCAPES    ADD TEXT  CHANGE TEXT

ABANDON    APPEND    MODIFY    .
BYE        INSERT    REPLACE   .
.          .         .         .
.          .         .         .

* HEL  APPEND⤸

APPEND    [FROM <SOURCE>...]]

          ADDS TEXT TO....
          .....
          .....
```

HELP is very handy when you're unsure of a command's syntax or capabilities.

Leaving SED with BYE

Type

```
* BYE⤸

Output file - :UDD:JACK:MYPROG.TOY
)
```

The BYE command terminates the editing session and updates your file by incorporating all your changes. Then it returns control to the CLI.

If you have changed an existing file, SED asks you if you want a backup file. For example, re-edit MYPROG.TOY:

```
) X  SED  MYPROG.TOY⤸

SED Rev ....
Input file ...

.       .
.       (SED displays text)

* DEL  LA⤸

.   .
.   (SED displays text)

* BYE⤸

Do you want to save the original file as a backup file?    Y⤸

Output file -   :UDD:JACK:MYPROG.TOY
Backup file is  :UDD:JACK:MYPROG.TOY.BU

)
```

The backup file is useful if you need to check the original contents of the file. For a source program backup file, you should rename the backup file from the CLI to have the appropriate ending; for example, add .F77 to the name.BU for a FORTRAN 77 source program.

SED allows only one backup file for any file; so, if a backup name.BU exists, and you tell SED to save the original as backup, SED will delete the older name.BU and replace it with the newer.

If you don't want a backup file, type N⤸ or simply ⤸ in response to the *...backup file?* question.

Summary

In this SED session, you've created and edited MYPROG.TOY, using commands XEQ, APPEND, LIST, MODIFY, INSERT, DELETE, FIND, POSITION, MOVE, DUPLICATE, SUBSTITUTE, DISPLAY, two function keys, and BYE. These commands will allow you to do nearly all the editing you want.

Three new text files result from the session: MYPROG.TOY, MYPROG.FOO (created by the SED MOVE command), and backup file MYPROG.TOY.BU. SED also created a file named MYPROG.TOY.ED for its own use, but you can ignore this.

Printing Files

You can print any text file with the CLI QPRINT command; e g.,

```
) QPRI  MYPROG.TOY ⤸
QUEUED, SEQ=n, QPRI=n
)
```

Now you can go to the line printer and pick up the printed file. (Operating the printer is described under "Printing Files" in Chapter 2.) You *can* print text from SED itself, but it's easier to do it from the CLI.

Session Summary

The following figure, Figure 4-1, shows all the commands you gave, with minimal text commentary. SED output isn't shown except where it relates directly to the preceding command.

A summary with short examples of each command follows the figure.

```
) DIR/I↓
) X  SED  MYPROG.TOY↓    (or SED
MYPROG.TOY↓)

SED   Rev. n    Input file -   MYPROG.TOY
Do you want MYPROG.TOY to be created?     Y↓

*                                          I
_____________________________

* APPEND↓

1          This is source line 1.↓
2          This is spurce line 2.↓
3          This is source line 3.↓

(ESC)

* LIST   ALL↓

1          This is source line 1.
2          This is spurce line 2.
3          This is source line 3.

* AP↓

4          This is line 4.↓
5          This is source line 6.↓

(ESC)

* MOD   2↓

2          This is spurce line 2.

    CTRL-F   CTRL-F   →   o↓

2          This is source line 2.

3          This is source line 3.

This was screen editing on line 2: two CTRL-Fs, → key,
then o↓.

↓

4          This is line 4.

    CTRL-F   CTRL-F   CTRL-E   source□   CTRL-E↓

4          This is source line 4.

5          This is source line 6.

(ESC)
```

Figure 4-1. SED Dialog Session (continues)

```
This was screen editing on line 4: CTRL-F, CTRL-F,
CTRL-E, then source□ then CTRL-E and ↓. (ESC)
terminated MODIFY mode. To help you keep track,
here is what the text looks like:

1          This is source line 1.
2          This is source line 2.
3          This is source line 3.
4          This is source line 4.
5          This is source line 6.

You started inserting text, then deleting it, here.

* IN   5↓

5          This is source line 5.↓

6

(ESC)

* DEL   6↓

5          This is source line 5.

* UNDO↓

6          This is source line 6.

* D   6↓

Command not unique, correct the command:

* DE   6↓

5          This is source line 5.

* F   '1.'↓

String not found, correct the command:

* F   '1.' ALL↓

1          This is source line 1.

* F   '2.'↓

String not found, correct the command:

* PO   1 ; F   '2.'↓

2          This is source line 2.

Just for review, the text looks like this:

1          This is source line 1.
2          This is source line 2.
3          This is source line 3.
4          This is source line 4.
5          This is source line 5.
```

Figure 4-1. SED Dialog Session (continued)

```
You started moving and duplicating text here:

* MOVE  2 BEFORE  1 )

1       This is source line 2.
2       This is source line 1.

* MOV  2 BEF  1 )

1       This is source line 1.
2       This is source line 2.

* MOV  1  3  ONTO MYPROG.FOO )

FILE   MYPROG.FOO   CREATED

1       This is source line 4.
2       This is source line 5.

* PO  1 )
* IN  FROM  MYPROG.FOO )

1       This is source line 1.
2       This is source line 2.
3       This is source line 3.
4       This is source line 4.
5       This is source line 5.

* DUP  2  5  ONT MYPROG.FOO )
.       .

* SUBS  'new source' for  'source' ALL )

1       This is new source line 1.
2       This is new source line 2.
3       This is new source line 3.
4       This is new source line 4.
5       This is new source line 5.

* DIS )

Edit file name -    :UDD:JACK:MYPROG.TOY

Current Page:           1
Current Line:           5

Last Page:              1
Last Line:              5

View range:             10
Display mode:           0

Typer mode:             ON
Blank mode:             OFF
Upper mode:             OFF

Line numbers being displayed
```

Figure 4-1. SED Dialog Session (continued)

```
* DUP AL ON MYPROG.FOO; DU A ON MYPROG.FOO)
.       .

* AP FR MYPROG.FOO ; AP FR MYPROG.FOO
.       .
.       .

Function keys: leftmost (first), then second, then second,
then first.

* HELP )

ESCAPES    ADD TEXT  CHANGE TEXT
ABANDON    APPEND    MODIFY
.          .         .

* HEL  APPEND )

APPEND    [FROM <SOURCE> ...]]

          ADDS TEXT TO....
          .....
          .....

* BYE )

Output file - :UDD:JACK:MYPROG.TOY

) XEQ SED MYPROG.TOY )

SED Rev ....
.       .
.       .

* DEL  LA )

* BYE )

Do you want to save the original file as a backup file?    Y)

Output file -   :UDD:JACK:MYPROG.TOY
Backup file is  :UDD:JACK:MYPROG.TOY.BU

) QPRINT MYPROG.TOY )
QUEUED, SEQ=n, QPRI=n
)

Go to printer and pick up printed file.
```

Figure 4-1. SED Dialog Session (concluded)

SED Command Summary

Table 4-1 summarizes the SED commands we've used. The *SED User's Manual,* 093-000249, describes all these commands in more detail.

What Next?

After the session and summaries, you have a working knowledge of the SED editor. You're ready to proceed to the language chapter of interest. (Or, if you wish, read about the other text editor, SPEED, in the next chapter).

Table 4-1. SED Commands Used in the Session

Action	Command Form	Example
Edit a file.	XEQ SED pathname or SED pathname	) X SED MYFILE ↵) SED MYFILE ↵
Add text to the file.	APPEND *[pathname]*	* APPEND↵ / PROGRAM MPROG↵ (ESC)
Examine what you have written.	LIST *[range]*	* LI 1 20 ↵
Change one or more lines.	MODIFY *[address]*	* MOD 2 ↵
Insert new lines of text.	INSERT	*IN 16 ↵ *16* This is a...↵ (ESC)
Delete one or more lines.	DELETE *[range]*	* DEL 18 20 ↵
Find a word or phrase.	FIND 'str' *[range]*	* F '(J.NE.0)' ALL ↵
Change position.	POSITION address	* PO -10 ↵
Move text within the file or onto another file.	MOVE *[range]* BEFORE address MOVE *[range]* AFTER address MOVE *[range]* ONTO pathname	* MOV 2 3 BEF LA ↵
Copy text within the file or onto another file.	DUPLICATE *[range]* BEFORE address DUPLICATE *[range]* AFTER address DUPLICATE *[range]* ONTO pathname	* DUP 1 3 BEF 10 ↵
Change many occurrences of a word or phrase to another word or phrase.	SUBstitute 'str' FOR 'str' *[range]*	* SUB 'Yes' for 'No' 1 20 ↵
Display edit status.	DISPLAY	* DIS ↵ *Edit file...*
Advance a screen.	Function key 2.	
Back up a screen.	Function key 1.	
Get help.	HELP *[keyword]*	* H ↵
Close and update file and return to the CLI.	BYE	* BYE↵)

End of Chapter

Chapter 5
Writing Text with the SPEED Editor

SPEED, like SED, is a text editor utility program supplied with your system. Like SED, SPEED lets you create and modify files containing upper- and lowercase ASCII text.

But using SPEED is very different from using SED. SPEED's commands are shorter and more cryptic than SED's. Unlike SED, SPEED can do arithmetic, execute commands conditionally, and edit invisible characters and delimiters like NEW LINE.

You can choose and use whichever editor you prefer. Either editor works with any text file.

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 we start the session.

To create and edit a file with SPEED, you must have append and write access -- as you always have for files within your user directory.

SPEED Features

SPEED is *character-string oriented.* This means you insert and edit character sequences at the current character position.

To indicate the current character position, SPEED displays a *character pointer* (CP). On display terminals, the CP appears as a blinking asterisk (*). (SED, the screen-oriented editor, emphasizes the current *line* position; SPEED, the character-oriented editor, emphasizes the current *character* position.)

To insert or edit text, you place the CP where you want, then do what you need to do. One SPEED command -- I -- serves to insert new text. Your I commands may include only one character or many lines of text.

SPEED Prompt and Delimiters

When it is ready for a command, SPEED displays an exclamation point (!) as a prompt. You then type a command and terminate it with CTRL-D. CTRL-D echoes as $$. For example, if you type the command 20L CTRL-D, it looks like 20L$$ as SPEED executes it.

In a change command, the ESC character serves as a text string delimiter. ESC 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 ESC as $ in this chapter.

CTRL-D echoes as $$ but *it is not* the same as two $$ characters; ESC echoes as $ but is not the same as one $ character. The $ character (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, then ignores the remainder of the command line.

SPEED Commands

The SPEED commands we will use, in the order we will use them, are listed in Table 5-1.

Table 5-1. Common SPEED Commands

Action	Command	Example
Edit the file.	XEQ SPEED pathname	) X SPEED/D MYFILE↵
Insert text in the file.	I	Ihello CTRL-D
Move to the beginning of a line.	L	20L CTRL-D
Jump to beginning or end of edit buffer.	J ZJ	J CTRL-D ZJ CTRL-D
Search for one or more characters.	S	Shello CTRL-D
Change one or more characters to other characters.	C	Chello$bye CTRL-D
Kill (delete) one or more lines of text.	K	2K CTRL-D
Type one or more lines of text.	T	10T CTRL-D
Get file status	F?	F? CTRL-D
Repeat (iterate) one or more commands.	<commands ;>	<C↵ $↵ ↵ $;> CTRL-D
File Update and Halt: include all edits to to the file, write it to disk, and return to the CLI.	FU$H	FU$H CTRL-D
File Backup and Halt: Same as FU, but also saves old file as a backup file.	FB$H	FB$H CTRL-D

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 buffer; the buffer isn't written to disk until you type FU or FB.

The edit buffer holds the entire current page (between form feed characters); if there are no form feeds in the file, the buffer holds the whole file. (SED also has an edit buffer, but you don't need to know about it to use SED effectively.)

Control Characters

SPEED does not use screen-edit/cursor control characters as the CLI and SED do. Instead, you use commands to set position and make editing changes. The CTRL characters for SPEED are

Character	What it Does
CTRL-I or TAB key	Produces a tab. Tabs are required in the syntax of several higher-level languages.
CTRL-S	Suspends display; useful for reading long files.
CTRL-Q	Continues display suspended by CTRL-S.
CTRL-U	Deletes the line you are typing.
CTRL-D	Terminates one or more SPEED commands. You can type CTRL-D at any time to get 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* (on display terminals) 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 is case insensitive, so, if your terminal has both lowercase characters, you can use either case at will for both commands and text. All *compilers* accept either case in text strings -- but some FORTRANs (not FORTRAN 77) and COBOL require *keywords* to be in uppercase. In the session, we use uppercase for SPEED commands and abbreviations; we use upper- and lowercase for the text itself.

If You Make Mistakes

SPEED generally protects your work from editing error, despite its terse error messages.

As you proceed through the session, it's okay to make your own mistakes -- everyone does, and they help teach. The main point is to learn the way SPEED works.

If at any time text seems to have vanished from the edit buffer, type FB$H CTRL-D to update the file and save the original version; then examine both the current and original versions with either SPEED or the CLI TYPE command and work with the one you want.

As with any program, CTRL-C CTRL-B will abort SPEED and return to the CLI. But CTRL-C CTRL-B nullifies all your edits as well, so use it only when you must.

Invoking SPEED

The execute command for SPEED has the form

) XEQ SPEED *//D/* pathname ⤸

The optional /D switch tells SPEED to display a range of lines around the CP and to show the CP. On a display terminal /D makes SPEED *much* easier to use, because SPEED shows your position after each command. On a printing terminal, omit the /D switch because printing the lines around the CP takes a lot of time.

In any case, if you omit /D, you must use T commands 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.

The pathname includes the name of the file you want to edit. Generally, you should use the DIR command to go into the directory you want before you execute SPEED; if so, the filename is the pathname.

After the XEQ command, SPEED will execute. If the file named in pathname already exists, SPEED will copy it into the edit buffer. If the file contains lowercase letters (as many text files do) SPEED will say **lowercase input encountered** as an informational message.

If the file named in pathname does not exist, SPEED will say

Create new file?

Type Y⤸ to have SPEED create the file; you will then begin editing with an empty edit buffer.

SPEED Session

To illustrate each SPEED command, we'll create and edit a prototype source program named MYPROG.TOY. (This is the same name used for SED, so, if you tried the SED chapter, use another filename, like MYPROG.SPEED.TOY .)

As we have done throughout this book, we'll show the prompt character (here, !) in **this typeface** in our examples. We'll show SPEED's output (aside from !) in *this typeface*. Type

```
) DIR/I ⤸
```

to ensure that your user directory is the working directory; then type

```
) X  SPEED/D  MYPROG.TOY ⤸
```

```
SPEED REV n
Create new file?  Y⤸
!
```

We received the *Create...* message because MYPROG.TOY did not exist before we ran SPEED; and we typed Y⤸ to have SPEED create the file.

Inserting New Text (I)

SPEED is displaying its ! prompt. Type the following, *precisely as shown* (but never include the ! prompt):

```
! 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, SPEED displays the text and the character pointer (CP). After each insertion, the CP points after the last inserted character -- here, 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 to add some text. Type

```
!This□is□source□line□3.⤸
CTRL-D
```

```
* Confirm?
```

This is typical of SPEED. We forgot the I command -- a common mistake -- before the new text. SPEED read our text as one of the commands that would take us back to the CLI -- losing all our edits. But SPEED asked for confirmation first. We don't want lose all our edits, so type

```
          N⤸
!
```

SPEED remains active because we answered N⤸ to the *Confirm?* question. If ever SPEED asks *Confirm?* and you don't want to lose your edits, type N⤸.

Let's include the I and try it again. Type

```
! IThis□is□source□line□3.⤸
CTRL-D
```

```
This is source line 1.
Thisis source line 2.
This is source line 3.
*
!
```

Forgetting I is a mistake that everyone makes. Having made it early, we'll remember it.

Because I is easy to forget, don't try to insert much text at once. To start, it's better to insert only a few lines, hit CTRL-D, and review what you've typed. Then continue inserting with a new I command.

Moving the CP to the Start of a Line (L)

Type

```
! -2L CTRL-D

This is source line 1.
*Thisis source line 2.
This is source line 3.
!
```

As you can see, L moves the CP around. With a negative number n, it moves the CP n NEW LINEs backward; with a positive number n it moves the CP n NEW LINEs forward; and without a number, L moves to the beginning of the current line. If n is out of range, L simply moves to the beginning of the first line or of the last line.

Type

```
! 1L CTRL-D
....SPEED displays text and CP

! - 400L CTRL-D
....SPEED displays text and CP

! 500L CTRL-D
....SPEED displays text and CP

!
```

Sequential 20L commands can help you read forward (in longer files); -20L commands help you read backward.

Jumping CP to Beginning or End of Buffer (J, ZJ)

J gets you to the start of the buffer; ZJ, to the end of the buffer.

Type this	Comment
! J CTRL-D	Go to start of buffer.
* *This is source line 1.*	SPEED does so.
....	
....	
! ZJ CTRL-D	Go to end of buffer.
....	
....	
This is source line 3.	SPEED does so.
*	
! IThis□is□line□4.↓ CTRL-D	Append text to buffer.
This is source line 1.	
....	
This is line 4.	Done.
*	
!	

As shown, ZJ is a convenient command when you want to append text to the file; you can simply start the insert after ZJ executes. (And J is easier to use than -nL to get you to the start of the buffer.)

Searching for Characters (S)

Type

! Sline□1. CTRL-D	Search for line 1.
Error:unsuccessful search	No luck.
Sline 1.$	
!	

SPEED searches from the CP onward, so the search failed -- because the CP was beyond the string we specified.

When a search (or change) command fails SPEED puts the CP at the beginning of buffer -- so an identical search should succeed. Type

! Sline□1. CTRL-D	Search for line 1.
This is source line 1. *	Found.
....	
!	

When a search (or change) command succeeds, the CP is after the last character sought. To search for ↓, type

! S↓	Search for ↓
CTRL-D	
**Thisis source line 2.*	Found.
....	
!	

As you saw, SPEED can find nonprinting characters like ↓. You'll see a use for this later in this session. As with any successful search, the CP is after the last character sought: after the ↓ on line 1, which is the beginning of line 2.

For a search (or change) to succeed, you must type in precisely the string you want; for example:

! Sline2. CTRL-D	Search for line2.
Error:unsuccessful search	Not found (because
....	we forgot the space in front of the 2).
!	

Searching for Characters (S) (continued)

SPEED does not allow a space between the S (or C) command letter and the string. If you insert one, SPEED won't be able to find the string. For example, type

```
! S□This CTRL-D                Search for □This
Error:unsuccessful search      No luck.
SThis$
!
```

By default, SPEED does not distinguish upper- from lowercase during searches. Type

```
! SLINE□1. CTRL-D              Search for LINE 1.
This is source line 1. *       Line 1. is found.
....
!
```

You'll be using S a lot, so try a few more:

```
! Shis CTRL-D                  Search for his#
....
....
! Ss□s CTRL-D                  Search for s□s
....
!
```

Changing a Character String (C)

C, like S, is a search command; but instead of simply searching, it searches and changes. C has the form Cstring1$string2 where string1 is the string you want to change to string2 and $ means pressing ESC. To *delete* a string, omit string2; e.g., type Cstring1 CTRL-D .

Let's use C to correct the typo in line 2:

```
! J CTRL-D                     To start of buffer.
! S1.⤶                         Position at start
CTRL-D                         of line 2.
....
*Thisis source line 2.         Done.
! Cis$□is CTRL-D               Change is to □is
Th □isis* source line 2.       Not what we wanted.
....
!
```

With C -- as with S -- you need to be accurate and specific. Let's fix it by typing

```
! J$CTh□isis$This□is CTRL-D     Combine J, C
                               commands.
....
This is* source line 2.        Success.
....
!
```

C can also extend changes over more than one line. For example, type

```
! Cline□4.⤶                    Change line 4.⤶
$line□4.⤶                      to line□4.⤶ and
This is source line 5.⤶        This is source line 5.⤶
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.         Done.
*
!
```

C -- like S -- is an extremely useful SPEED command. It allows you to edit without pinpointing the CP with search, insert, or delete commands.

Deleting Lines (K)

The C command allows you to delete characters and lines; but K is much more convenient. nK deletes n lines forward for a positive n or n lines backward for a negative n. (A *line* is the characters between the CP and the next NEW LINE character.)

To display the text, type

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)              Get to the right line.

* This is source line 3.       Done.
....
! 1K CTRL-D                    Kill it.

....
This is source line 2.         Done.
* This is line 4.
....
! IThis is source line 3.⤶     Reinsert
CTRL-D                         it.
....
This is source line 3.         Done.
....
!
```

To delete *the remainder* of a line, simply put the CP before the doomed string, and type 1K. Type

```
! S5$L$SThis CTRL-D          Position before string.
....
This* is source line 5.      Done.
....
! 1K CTRL-D                  Kill rest of line.
....
This*                        Done.

! I□is□source□line□5.J       Now restore string...
CTRL-D
....
This is source line 5.       Done.
*
!
```

Often it's easier to delete part or all of a bad line and insert a new one than to try to correct the original. As you saw, K can help you do this.

With K, we suggest that you use only positive values of n to keep your editing simple. A negative n when the CP is in the middle of a line will delete not only the previous line but also the left portion of the current line.

At this point, you've executed SPEED, 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 restored 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.

The following commands simply make editing easier.

Typing lines (T)

Type

```
! J$T CTRL-D
* This is source line 1.
!
```

T types one or more lines on the terminal. Because the /D switch instructs SPEED to type lines around the CP, and to show the CP, you rarely need to use T directly. But still, T is useful when you want to see a different range of lines than the /D default, lines far away from the CP, or when you want to type the entire buffer. If you *didn't* use the /D switch, you'll use T extensively to see what's going on.

When you use T, SPEED types only the specified number of lines. It shows the CP only if you omit an argument; for example, if you type 20T CTRL-D, SPEED will not show the CP. The most common forms of T are

#T	Type the entire buffer.
T	Type the current line and the CP.
nT	Type n lines, including current line.
0,nT	Type buffer from its beginning to character n.

For example, type

```
! #T CTRL-D
 ....
 ....

! 2T CTRL-D
 ....
 ....
```

Getting File Status (F?)

Type

```
! F? CTRL-D

Global:
    Input File -        :UDD:JACK:MYPROG.TOY
    Output File -       :UDD:JACK:MYPROG.TOY.TM
    Update Mode         On
Local:
    Input File -        None
    Output File -       None

!
```

The F? command describes some things we don't explain here, but it can help you keep track of the file you're editing.

Repeating (Iterating) 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 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.

Repeating (Iterating) Commands (<...>) (continued)

For example, type

```
! J CTRL-D                    Start of buffer.
....
....

! <C↓                         Repeat <Change↓
$ ↓                           to ↓ and
↓                             ↓
$;> 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 () in the buffer to two NEW LINES (↓) -- producing 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 -- and you can do it this easily only in SPEED. (You can't do it with a SED SUBSTITUTE command.)

File Update, File Backup and Halt (FU$H or FB$H)

Type

```
! FU$H CTRL-D

)
```

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 to the CLI. FB$H does the same thing but saves the original file as backup, adding the characters .BU to the filename.

In our example session, saving the original file would be pointless because we created it during this session.

Try re-editing MYPROG.TOY:

```
) X  SPEED/D  MYPROG.TOY↓
```

**This is source line 1.*

This is source line 2.

This is source line 3.

This is line 4.

This is source line 5.

```
! CThis□is$This□is□new□ CTRL-D
```

This is new source line 1.*

```
....

! FB$H CTRL-D

)
```

Because you specified a backup file, there will be both a MYPROG.TOY and a MYPROG.TOY.BU in the working directory -- the latter will be in the backup file.

Typing XEQ SPEED/D pathname each time is a nuisance. You might want to ask your system manager to create a macro -- named something like SPEED.CLI, in :UTIL -- that would allow you to type simply SPEED pathname.

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 RENAME the backup file from the CLI to have the appropriate ending; for example, add .F77 to the name.BU for a FORTRAN 77 source program.

SPEED allows only one backup file for any file; so, if a backup name.BU exists, and you type FB$H, SPEED will delete the older name.BU and replace it with the newer name.BU.

Actually, you can type either the FU or FB commands without H, but since the commands clear the buffer, you'd have to type H anyway to get back to the CLI and re-exectute SPEED. So you might as well say 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 H 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 haven't changed the file. You might also exit with H if you realize that you made substantial errors and you want to start again.

Finally, we'll tell you that you can *omit* the ESC ($) delimiter after all but insert, search, and change commands. For example, JS1. CTRL-D and FUH CTRL-D are legal.

Summary

In this SPEED session, you've created and edited MYPROG.TOY, using commands XEQ, L, J, S, C, K, F?, <...>, 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: MYPROG.TOY and MYPROG.TOY.BU (created by the SPEED FB$ command).

Session Summary

The following figure, Figure 5-1, shows all the commands you gave, with minimal text commentary. SPEED output isn't shown except where it relates directly to the preceding command.

A summary with short examples of each command follows the figure.

```
) DIR/I )
) X  SPEED/D  MYPROG.TOY )
SPEED REV n
Create new file?  Y)

Using the I, L, and J Commands

! IThis□is□source□line□1.)
Thisis□source□line□2.)
CTRL-D

This is source line 1.
Thisis source line 2.
*

! This is source line 3.)
CTRL-D

* Confirm?    N)

! IThis□is□source□line□3.)
CTRL-D

This is source line 1.
Thisis source line 2.
This is source line 3.
*

! -2L CTRL-D

This is source line 1.
* Thisis source line 2.
This is source line 3.

! 1L CTRL-D
...SPEED displays text and CP
```

Figure 5-1. Summary of SPEED Session (continues)

```
! - 400L CTRL-D
...SPEED displays text and CP

! 500L CTRL-D
...SPEED displays text and CP

! J CTRL-D                          Go to start of buffer.
* This is source line 1.           SPEED does so.
....
....

! ZJ CTRL-D                         Go to end of buffer.
....
....
This is source line 3.             SPEED does so.
*

! IThis□is□line□4.)                Append text to buffer.
CTRL-D

This is source line 1.
....
This is line 4.                    Done.
*
!

Using the S Command

! Sline□1. CTRL-D                   Search for line 1.
Error:unsuccessful search          No luck.
Sline 1.$

! Sline□1. CTRL-D                   Search for line 1.
This is source line 1. *           Found.
....

! S)                                Search for )
CTRL-D
* Thisis source line 2.            Found.
....

! Sline2. CTRL-D                    Search for line2.
Error:unsuccessful search          Not found (because
...                                we forgot the space).

! S□This CTRL-D                     Search for □This
Error:unsuccessful search          No luck.
STHIS$

! SLINE□1. CTRL-D                   Search for LINE 1.
This is source line 1. *           Line 1. is found.
....

! Shis CTRL-D                       Search for his
....
....
! Ss□s CTRL-D                       Search for s□s
....
!
```

Figure 5-1. Summary of SPEED Session (continued)

Using the C Command

```
! J CTRL-D                          To start of buffer.
! S1↓                               Position at start
CTRL-D                              of line 2.
....
*Thisis source line 2.              Done.
! Cis$□is CTRL-D                    Change is to □is
Th □isis* source line 2.            Not what we wanted.
....

! J$CTh□isis$This□is CTRL-D                Combine J, C
                                            commands.
....
This is* source line 2.                     Success.
....

! Cline□4.↓                         Change line 4.↓
$line□4.↓                           to 4.↓ and
This is source line 5.↓             This is source line 5.↓
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.              Done.
*
!
```

Using the K Command

```
          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.
*

! J$S3$L CTRL-D                     Get to the right line.

* This is source line 3.            Done.
....
! 1K CTRL-D                         Kill it.

....
This is source line 2.              Done.
* This is line 4.
...
! IThis is source line 3.↓          Reinsert
CTRL-D                              it.

...
This is source line 3.              Done.
...
```

Figure 5-1. Summary of SPEED Session (continued)

```
! S5$L$SThis CTRL-D                 Position before string.
....
This* is source line 5.             Done.
....
! 1K CTRL-D                         Kill rest of line.
....
This*                               Done.

! I □is□source□line□5.↓            Now restore string...
CTRL-D
....
This is source line 5.              Done.
*
```

Using the T Command

```
! J$T CTRL-D
* This is source line 1.

! #T CTRL-D
....

! 2T CTRL-D
....
....
```

Using the F? Command and Repeating Commands

```
! F? CTRL-D

Global:
  Input File-           :UDD:JACK:MYPROG.TOY
  Output File-          :UDD:JACK:MYPROG.TOY.TM
    Update Mode         On
Local:
      Input File -      None
      Output File -     None

! J CTRL-D                          Start of buffer.
....
....

! <C↓                               Repeat <Change↓
$↓                                  to↓ and
↓                                   ↓
$;> 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.
```

Figure 5-1. Summary of SPEED Session (continued)

Using the FU$H and FU$B Commands

```
! FU$H CTRL-D
) X  SPEED/D  MYPROG.TOY )
This is source line 1.
This is source line 2.
This is source line 3.
This is line 4.
This is source line 5.
! CThis□is$This□is□new□ CTRL-D
This is new source line 1.
....
! FB$H CTRL-D
)
```

Figure 5-1. Summary of SPEED Session (concluded)

SPEED Summary and Review

Table 5-2 summarizes the SPEED commands we've used. For details on these and SPEED's many other commands read the *AOS and AOS/VS SPEED Text Editor User's Manual,* 093-000197.

What Next?

After the session and summaries, you have a working knowledge of the SPEED editor. You're ready to proceed to the language chapter of interest.

Table 5-2. SPEED Command Examples

	Example(s)	Meaning and Result
XEQ SPEED	X SPEED/D FILE1)	Execute SPEED with /Display; if FILE1 exists, read it into edit buffer, otherwise offer to create it.
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(B.NE.0)$T CTRL-D	Search for the characters (B.NE.0) and, if you find them, type the line.
C	CZZ9$ZZZ9$T CTRL-D	Search for characters ZZ9 and, if found, change to ZZZ9 ; type line.
L	L$I) CTRL-D	Insert a NEW LINE () character before the current line.
T	#T CTRL-D	Type entire buffer. current line.
K	3K CTRL-D	Kill (delete) three lines.
F?	F? CTRL-D	Get file status.
< $;>	J$<CNo$Yes$;>CTRL-D	From start of buffer, change every Yes to No .
FUH and FBH	FUH CTRL-D	File update: write edited file to disk.

End of Chapter

Chapter 6
Instant FORTRAN Programming

This chapter covers three DG FORTRANs; FORTRAN IV, FORTRAN 5, and FORTRAN 77 (nicknamed F77). There is one program example for FORTRAN IV and FORTRAN 5, another for F77.

These are the steps you follow to create a program in any of these FORTRANs:

1. Create or edit a FORTRAN source file with SED or SPEED:

   ```
   ) XEQ  SED  pathname ↵
      or
   ) XEQ  SPEED/D  pathname ↵
   ```

   Via your chosen editor, type in the FORTRAN statements and comments that make up the program.

2. Compile the source file with the CLI command

   ```
   ) FORT4  pathname ↵
      or
   ) F5  pathname ↵
      or
   ) F77  pathname ↵
   ```

   For F77, the compile line can also include the /DEBUG switch, described later.

3. If there are any compilation errors, return to step 1 and fix the offending statement(s). If there are no compilation errors, go to step 4.

4. Link the object file to produce an executable program:

   ```
   ) XEQ LINK  pathname [subprograms] FORT.LB↵
      or
   ) F5LD  pathname [subprograms] ↵ (FORTRAN 5)
      or
   ) F77LINK  pathname [subprograms] ↵      (F77)
   ```

   For F77, the F77LINK line can also include the /DEBUG switch, described later.

5. Execute the program with the CLI command:

   ```
   ) XEQ  pathname ↵
   ```

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. For F77, if you included the /DEBUG switches and your system has the SWAT™ interactive debugger, debug the program via SWAT pathname↵.

8. Go to step 1 and fix the erroneous statement(s).

9. You're done!

This chapter guides you through all the steps you need to write and execute FORTRAN IV/FORTRAN 5 and F77 programs.

The FORTRAN Example Programs

There are two example programs: one for FORTRAN IV/FORTRAN 5, and one for F77. The two do exactly the same thing; only the syntax differs slightly.

Each example program is a simple program to calculate home mortgage payments; it produces a schedule of monthly principal and interest. Each program uses only ordinary arithmetic operations, calls no subroutines, and (excluding comments) is only about half a page long. Each program uses two formulas. You need not understand how these formulas work to understand the illustrations.

The mortgage formulas are those used by banks within the United States. Different formulas are used outside the US: in Canada, Europe, and Asia. So, if you live outside the US, treat the formula parts of the program as an example. (Later, you might want to replace the existing formulas with your national formulas.)

Figure 6-1 shows a flowchart for both programs. Figure 6-2 shows the initial version of the FORTRAN IV/FORTRAN 5 program, MORTGAGE.FR, complete with errors. Figure 6-3 shows the initial version of the F77 program, MORTGAGE.F77, complete with errors.

Even if you decide not to try one of these programs, you should at least examine Figures 6-1 and either 6-2 or 6-3 before proceeding to the next section.

To compile and run FORTRAN programs, your system must have the appropriate compiler files and libraries. The directory that holds these -- often directory :UTIL -- must be in your search list.

Writing the FORTRAN Program

If you want to try a 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 ↵
) CREA/DIR  FORT↵
) DIR  FORT↵
)
```

Now, use SED or SPEED to create the source file. For FORTRAN IV/FORTRAN 5, use the name MORTGAGE.FR; for F77, use the name MORTGAGE.F77.

```
) XEQ  SED  MORTGAGE      { .FR↵ }
                          { .F77↵ }
   or
) XEQ  SPEED/D  MORTGAGE  { .FR↵ }
                          { .F77↵ }
```

Type in the program according to Figure 6-2. Don't forget to insert a tab (use the TAB key or CTRL-I) before each FORTRAN statement. The FORTRAN IV and FORTRAN 5 compilers require all *statements* to be in uppercase. The F77 compiler doesn't care about case.

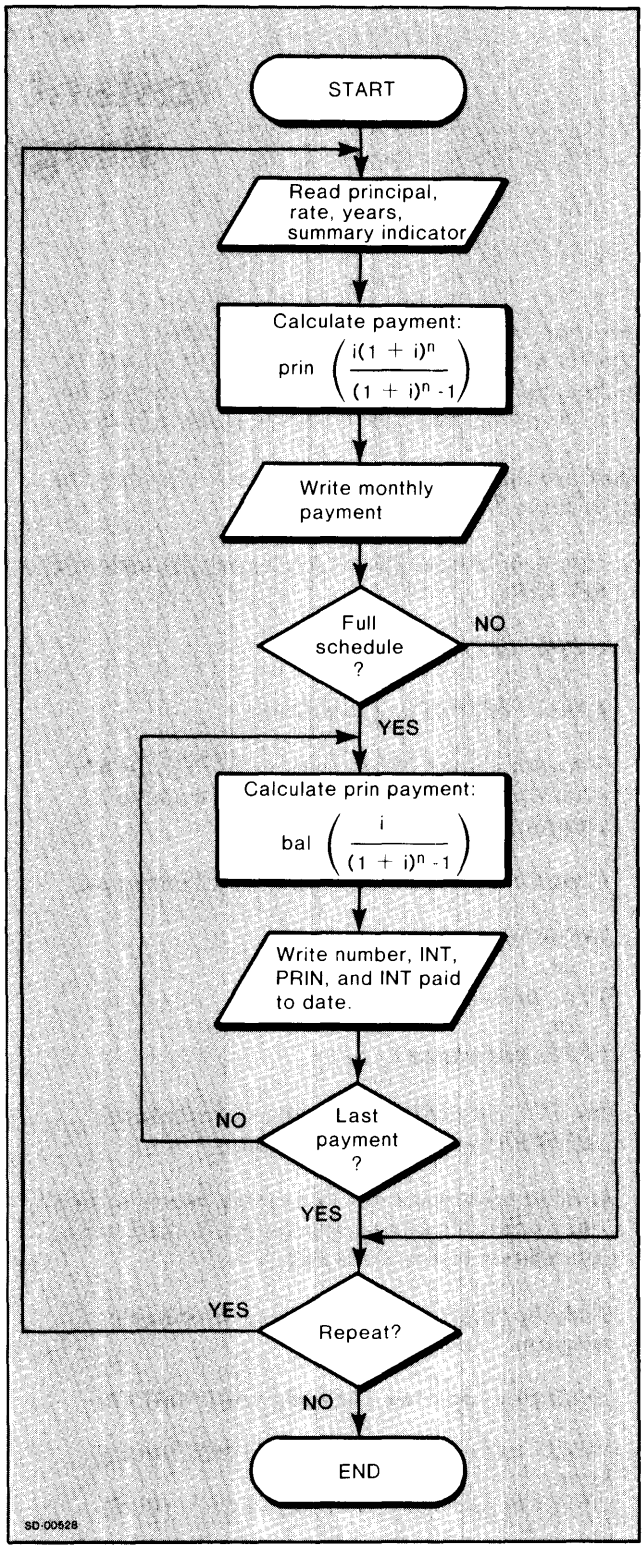


Figure 6-1. MORTGAGE.FR Program Flowchart

```
C   This F4/F5 program computes mortgage payments: summary or full schedule.
C       Declare double precision variables to handle today's rates.
        DOUBLE PRECISION  AMOUNT, RATE, R, PAY, BAL
        DOUBLE PRECISION  INTEREST, ITD, PRIN

5       TYPE "Enter amount, rate, years, and 0 for summary or 1"
        TYPE "for full schedule. Separate entries with a comma, end"
        TYPE "input with NEW LINE; e.g., 60000, .14, 25, 0 [NL]"
        ACCEPT  AMOUNT, RATE, IYEARS, IFULL   ; Get figures from terminal.

C  Change yearly rate RATE to monthly rate R.
        R = RATE/12
C  Change years IYEARS to months MONTHS.
        MONTHS = IYEARS*12

C  Calculate monthly payment PAY, write header and PAY.
        PAY = AMOUNT*R*(1+R)**MONTHS   /  ((1+R)**MONTHS -1)
        WRITE (10,110) AMOUNT, RATE, IYEARS, PAY
110     FORMAT (1H0, "Amount    = $", F9.2 ,/, " Interest Rate =",F7.4,/,
     X  " Loan Life is ",I3," Years",/, " Monthly Payment = $", F9.2, /)

C   If user specified summary, skip full schedule.
        IF (IFULL .LE. 0) GO TO 400

C   Full schedule -- set up variables.
        BAL = AMOUNT    ; Initial Loan Balance equals original amount.
        MM = MONTHS     ; Save original number of months in MM.
        ITD = 0         ; Interest To Date is initially 0.

C   To LISTFILE, write header, compute and write figures month by month.
        WRITE (12,110) AMOUNT, RATE, IYEARS, PAY
        WRITE (12,120)
120     FORMAT (1X," Num",8X,"Interest",6X,"Prin. Pay",6X,"Prin. Bal."
     X  4X,"Interest Paid to Date",/)

        DO 200  I=1, MM  ; DO until you reach MM months...
C          Calculate amount of principal in payment, PRIN.
           PRIN = BAL*R / ((R+1)**MONTHS-1)
C          Decrement month.
           MONTHS = MONTHS -1
C          Calculate amount of interest in payment, INTEREST.
           INTEREST = PAY-PRIN
C          Update loan balance.
           BAL = BAL-PRIN
C          Update interest paid to date.
           ITD = ITD+INTEREST
C          Write figures for this month.
           WRITE (12,140) I, INTEREST, PRIN, BAL, ITD
140        FORMAT (1X,I3, 7X,"$",F9.2, 5X,"$",F9.2, 5X,"$",F9.2, 8X,"$",F9.2,/)
200     CONTINUE

400     ACCEPT " Type 1 to repeat, 0 to stop. ", ISTOP
        IF (ISTOP .GT. 0) GO TO 5
        END
```

Figure 6-2. FORTRAN IV/FORTRAN 5 MORTGAGE Program with Errors

```
C   This F77 program computes mortgage payments: summary or full schedule.
C       Declare double precision variables to handle today's rates.
        DOUBLE PRECISION  AMOUNT, RATE, R, PAY, BAL
        DOUBLE PRECISION  INTEREST, ITD, PRIN

5       PRINT *, "Enter amount, rate, years, and 0 for summary or 1"
        PRINT *, "for full schedule. Separate entries with a comma, end"
        PRINT *, "input with NEW LINE; e.g., 60000, .14, 25, 0 [NL]<NL>"
        READ *,  AMOUNT, RATE, IYEARS, IFULL   ! Get figures from terminal.

C  Change yearly rate RATE to monthly rate R.
        R = RATE/12
C  Change years IYEARS to months MONTHS.
        MONTHS = IYEARS*12

C  Calculate monthly payment PAY, write header and PAY.
        PAY = AMOUNT*R*(1+R)**MONTHS   /  ((1+R)**MONTHS -1)
        PRINT 110,  AMOUNT, RATE, IYEARS, PAY
110     FORMAT (1H0, "Amount    = $", F9.2 ,/, " Interest Rate =",F7.4,/,
     X  " Loan Life is ",I3," Years",/, " Monthly Payment = $", F9.2, /)

C   If user specified summary, skip full schedule.
        IF (IFULL .LE. 0) GO TO 400

C   Full schedule -- set up variables.
        BAL = AMOUNT     ! Initial Loan Balance equals original amount.
        MM = MONTHS      ! Save original number of months in MM.
        ITD = 0          ! Interest To Date is initially 0.

C   To LISTFILE, write header, compute and write figures month by month.
        WRITE (12,110) AMOUNT, RATE, IYEARS, PAY
        WRITE (12,120)
120     FORMAT (1X," Num",8X,"Interest",6X,"Prin. Pay",6X,"Prin. Bal.",
     X  4X,"Interest Paid to Date",/)

        DO 200  I=1, MM  ! DO until you reach MM months...
C          Calculate amount of principal in payment, PRIN.
           PRIN = BAL*R / ((R+1)**MONTHS-1)
C          Decrement month.
           MONTHS = MONTHS -1
C          Calculate amount of interest in payment, INTEREST.
           INTEREST = PAY-PRIN
C          Update loan balance.
           BAL = BAL-PRIN
C          Update interest paid to date.
           ITD = ITD+INTEREST
C          Write figures for this month.
           WRITE (12,140) I, INTEREST, PRIN, BAL, ITD
140        FORMAT (1X,I3, 7X,"$",F9.2, 5X,"$",F9.2, 5X,"$",F9.2, 8X,"$",F9.2,/)
200     CONTINUE

400     PRINT *, "Type 1 to repeat, 0 to stop. '
        READ *, ISTOP
        IF (ISTOP .GT. 0) GO TO 5
        END
```

Figure 6-3. FORTRAN 77 MORTGAGE Program with Errors

Compiling with FORTRAN IV and FORTRAN 5

Now we can compile our initial version of MORTGAGE.FR. (For FORTRAN 77, skip to the next section.) Since the initial version usually has some syntactical errors, we use the /NA command switch with the compile command. This switch tells the compiler to scan the input for errors, but it produces no output file (producing one would do no harm, but it would take longer). The compile command is

```
) FORT4/NA   MORTGAGE ↵      For FORTRAN IV.
or
) F5/NA   MORTGAGE ↵         For FORTRAN 5.
```

For FORTRAN IV, the compiler says

```
;          IF (ISTOP .GT. 0) GO TO 5
; *** 111 *** CHR 01
```

This means that error number 111 occurred on *or before* character position 1 in the IF (ISTOP..) statement. Looking at Appendix B of the *FORTRAN IV User's Manual,* we see that error 111 means "Hollerith constant not ended at statement end". The error *actually* occurred on the preceding line, where we started the text string with a quotation mark (") and ended it with an apostrophe (').

If we are using FORTRAN 5, the compiler announces the date and time, and then says

```
LINE 52, UNCLOSED STRING CONSTANT IN
     STATEMENT

1 COMPILE ERROR
```

FORTRAN 5 is pretty specific; it pinpoints line 52, where we started a string constant with a quotation mark (") and ended it with an apostrophe (').

In their different ways, both compilers have told us about the same error: starting a string constant with " and ending it with '. Using SED or SPEED, we correct the line, producing:

```
400    ACCEPT "Type 1 to repeat,0 to stop. ",ISTOP
```

After making these changes, repeat the compile command, this time omitting the /NA global switch:

```
) FORT4   MORTGAGE ↵
   or
) F5   MORTGAGE ↵
```

This time, we receive no error messages. The CLI's) prompt returns, indicating that we can now begin the next step in the procedure: linking with the system Link utility. This is described *after* the next section.

Compiling with FORTRAN 77

Now we can compile our initial version of MORTGAGE.F77. Since the initial version usually has some syntactical errors, we use the /N switch so that we'll receive error messages.

```
) F77/N   MORTGAGE ↵
```

The compiler announces itself, the date, time, and some options selected by the compile macro. Then it says

```
ERROR 255 SEVERITY 2 BEGINNING ON LINE 52
Missing string delimiter in a character constant.
```

FORTRAN 77 error messages are very specific. This one pinpoints line 52, where we started a character constant with a quotation mark (") and ended it with an apostrophe ('). Using SED or SPEED, we correct the line, producing:

```
400      PRINT *, "Type 1 to repeat, 0 to stop. "
```

After making this change and leaving the editor, we issue the F77 MORTGAGE↵ command again, omitting the /N switch. Compiling finishes without error messages, and we receive the CLI's) prompt. We can now begin the next step: linking with the system's Link utility.

Creating the Program File with Link

The Link command line for MORTGAGE is

```
) XEQ LINK   MORTGAGE   FORT.LB ↵   (for FORTRAN
IV)
   or
) F5LD   MORTGAGE ↵   (for FORTRAN 5)
   or
) F77LINK   MORTGAGE ↵      (for F77)
```

As Link builds the MORTGAGE program file, it says

```
LINK REVISION nnnn ON date AT time
=MORTGAGE.PR   CREATED
```

In FORTRAN, or any other higher-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 *Link User's Manual.*

Executing the FORTRAN Program

We can proceed to the next step, executing MORTGAGE.PR. To execute this or any other program, simply type the XEQ command and follow it with the program name and ↵.

```
) XEQ   MORTGAGE ↵
```

Enter amount, rate, years, and 0 for summary or 1 for full schedule. Separate entries with a comma, end input with NEW LINE; e.g., 60000, .14, 25, 0 [NL]

We then respond with a request for the summary information (monthly payment only), given a mortgage of $60,000 at 14% for 25 years:

```
60000,  .14,  25,  0 ↵
```

MORTGAGE then says

Amount = $60000.00
Interest Rate = 0.1400
Loan Life is 25 Years
Monthly Payment = $ 722.26

Type 1 to repeat, 0 to stop.

To continue, type

```
1↵
```

Since we responded with 1, the same instructions appear on the screen. This time, enter the same arguments but ask for a full schedule:

```
60000, .14, 25, 1 ↵
```

Amount = $60000.00
Interest Rate = 0.1400
Loan Life is 25 Years
Monthly Payment = $ 722.26

..... ERROR nnn ...
CALLED .... nnnn

A fatal runtime error! And we find the CLI running on the terminal. FORTRAN IV doesn't identify the error but reports it as *ERROR 14* (the FORTRAN IV manual describes *Runtime Error 14* as an "I/O error." FORTRAN 5 and F77, as usual, are more explicit, saying that *FILE DOES NOT EXIST.*

The source of our problem is in the *preconnection* to unit 12. DG FORTRANs all have preconnections between the following units and files:

Unit Number	File
9	@DATA
10	@OUTPUT
11	@INPUT
12	@LIST

@DATA, etc., and @LIST are actually pointers to filenames, called *generic files.* As described in Chapter 3 under LISTFILE, these 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, the system couldn't find the @LIST file and aborted the program with a runtime error message.

To correct this, we simply set the list file to the terminal:

```
) LISTFILE   @CONSOLE↵
```

Having set the list file to the terminal, we re-execute the program; then we give the same figures, asking for a full schedule:

```
) X  MORTGAGE ↵
```

Enter amount....
....

```
60000, .14, 25, 1 ↵
```

Amount = $ 60000.00
Interest Rate = .1400
Loan Life is 25 Years
Monthly Payment = $ 722.26

Num	*Interest*	*Prin.Pay*	*Prin. Bal.*	*Interest...*
1	*$700.00*	*$22.26*	*$59977.74*	*$700.00*
2	*$699.74*	*$22.52*	*$59955.23*	*$1399.74*
.	.	.	.	.

It works! Let's 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, execute it again, and give the same figures:

```
) LISTF   MORTGAGE.OUT ↵
) XEQ   MORTGAGE↵
```
Enter amount....
....

```
60000, .14, 25, 1↵
```

Amount   = $ 60000.00
.

We see the same summary figures as before, then wait a moment as MORTGAGE writes the full schedule to file MORTGAGE.OUT. Then MORTGAGE says and we reply

Type 1 to repeat, 0 to stop. 0↵
STOP
)

MORTGAGE has terminated, closing list file MORTGAGE.OUT. MORTGAGE.OUT 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 it, type

) QPRI MORTGAGE.OUT ↵
QUEUED, SEQ=n, QPRI=n
)

Checking the printer, we find the full schedule, part of which is shown in Figure 6-4. The entire schedule is eight or so pages long. The interest total at the end (which the Figure 6-4 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    = $ 60000.00
Interest Rate = 0.1400
Loan Life is  25 Years -
Monthly Payment = $   722.26

 Num        Interest       Prin. Pay      Prin. Bal.     Interest Paid to Date
  1       $   700.00     $    22.26     $ 59977.74       $   700.00
  2       $   699.74     $    22.52     $ 59955.23       $  1399.74
  3       $   699.48     $    22.78     $ 59932.45       $  2099.22
  4       $   699.21     $    23.04     $ 59909.40       $  2798.43
  5       $   698.94     $    23.31     $ 59886.09       $  3497.37
  6       $   698.67     $    23.59     $ 59862.50       $  4196.04
  7       $   698.40     $    23.86     $ 59838.64       $  4894.44
  8       $   698.12     $    24.14     $ 59814.50       $  5592.56
  9       $   697.84     $    24.42     $ 59790.08       $  6290.39
 10       $   697.55     $    24.71     $ 59765.38       $  6987.94
 11       $   697.26     $    24.99     $ 59740.38       $  7685.21
 12       $   696.97     $    25.29     $ 59715.10       $  8382.18
 13       $   696.68     $    25.58     $ 59689.52       $  9078.85
 14       $   696.38     $    25.88     $ 59663.64       $  9775.23
 15       $   696.08     $    26.18     $ 59637.46       $ 10471.31
 16       $   695.77     $    26.49     $ 59610.97       $ 11167.08
 17       $   695.46     $    26.80     $ 59584.18       $ 11862.54
 18       $   695.15     $    27.11     $ 59557.07       $ 12557.69
 19       $   694.83     $    27.42     $ 59529.64       $ 13252.52
 20       $   694.51     $    27.74     $ 59501.90       $ 13947.03
 21       $   694.19     $    28.07     $ 59473.83       $ 14641.22
 22       $   693.86     $    28.40     $ 59445.44       $ 15335.08
```

Figure 6-4. Start of Full Schedule from MORTGAGE Program (FORTRAN)

Remember the List File

The list file is still set to MORTGAGE.OUT . But the next time you log on, it will be set to the dummy filename @LIST, as when we executed MORTGAGE. So, if you later execute MORTGAGE, ask for a full schedule, and get a *FILE DOES NOT EXIST* error, just set the list file to @CONSOLE or a disk filename.

You can see the flexibility of the list file, preconnected to unit 12. It allows us to set our output file at will from the CLI.

On Using the SWAT™ Debugger

With F77, DG's SWAT™ debugger can really ease the debugging phase of F77 program development. To use SWAT, compile your program with the /DEBUG switch and ask for a listing with the /L=@LPT switch:

```
) F77/DEBUG/L=@LPT   pathname ↵
```

Link using the /DEBUG switch (this cannot be abbreviated):

```
) F77LINK/DEBUG   pathname ↵
```

Start up the program in SWAT:

```
) SWAT   pathname ↵
 ...SWAT  Rev...
>
```

In SWAT, you can: set breakpoints by listing line number via the BREAKPOINT command; list source lines to check logic; start or run the program with CONTINUE; examine variables at breakpoints with TYPE; get help at any point; and leave SWAT with BYE.

For example, a SWAT session with MORTGAGE might involve the following dialog:

```
) F77/DEBUG/L=@LPT   MORTGAGE ↵
.
) F77LINK/DEBUG   MORTGAGE ↵
.
) SWAT MORTGAGE ↵
.
> BREAK 21 ↵
Set at .MAIN. 21

> CON ↵
Enter amount...
60000, .14, 25, 0 ↵

Breakpoint trap at .MAIN. 21.

> TYPE PAY ↵
7.222566...+.02
> BYE ↵
Swat terminated
)
```

Summary

With the correct listing (and, for F77, the SWAT information), you have completed the FORTRAN chapter -- and have acquired a sound basis for programming with DG FORTRANs.

What Next?

If you want to try another language, proceed to the appropriate chapter; or, if you want, review earlier material. Or, you can start writing your own programs, using the other pertinent DG FORTRAN manuals.

End of Chapter

Chapter 7
Instant COBOL Programming

These are the steps you follow to write a COBOL program:

1. Write and, if necessary, edit the COBOL program source file with one of the editors: SED or SPEED/D.

) XEQ SED pathname ⤸
   or
) XEQ SPEED/D pathname ⤸

2. Compile the source file with the CLI command

) COBOL/L pathname ⤸

3. If there are any compilation errors, go to step 1 and fix them with your chosen editor.

4. Build the object file into an executable program file with the CLI command

) CBIND pathname⤸

5. Execute the program with the CLI command

) XEQ pathname ⤸

6. If the program runs the way you want it to, go to step 9.

7. Identify logic errors via runtime error messages or erroneous output (COBOL has its own debugger, but this is not covered in this primer).

8. Go to step 1 and fix the incorrect code with your chosen text editor.

9. You're done!

The COBOL Example Program

The program in this chapter, like the FORTRAN program in Chapter 6, calculates home mortgage payments. It can also produce a month-by-month schedule of principal and interest. Figure 7-1 is the program flowchart and Figure 7-2 is the COBOL program itself.

The mortgage formulas are those used by banks within the United States. Different formulas are used outside the US: in Canada, Europe, and Asia. So, if you live outside the US, treat the formula parts of the program as an example. (Later, you might want to replace the existing formulas with your national formulas.)

If you tried the FORTRAN program, you'll see startling differences between it and the COBOL program. The COBOL program defines all variables ("data names") and record structure at the beginning; and it has divisions, sections, sentences and clauses. It is much more structured and much longer than its FORTRAN counterpart.

To compile and run COBOL programs, your system must have the appropriate compiler files and libraries. The directory that holds these, (often directory :UTIL) must be in your search list.

Even if you decide not to try the COBOL program, you should at least examine Figures 7-1 and 7-2 before proceeding.

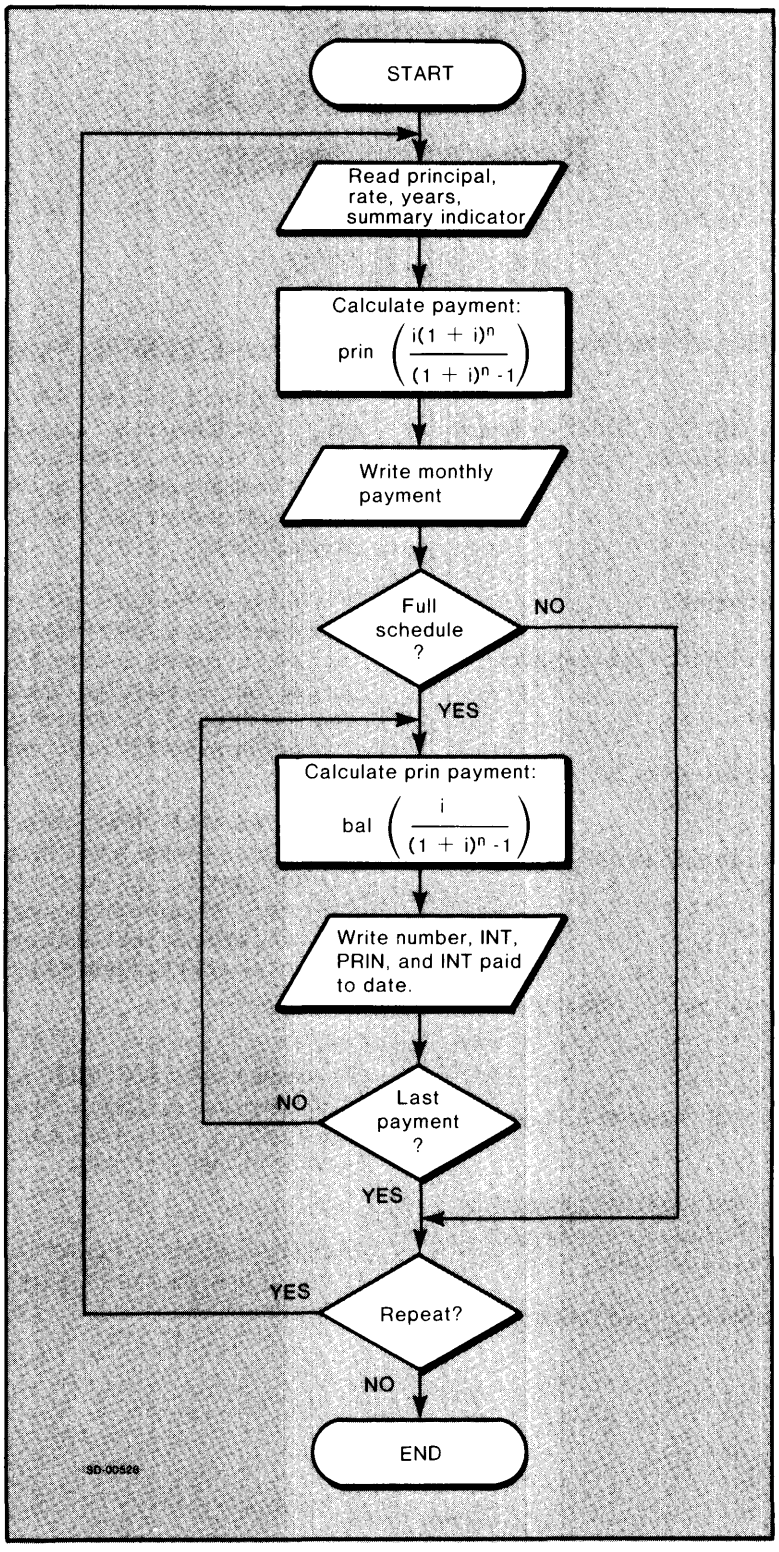


Figure 7-1. MORTGAGE Program Flowchart

```
    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).
    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.
```

Figure 7-2. COBOL MORTGAGE Program (continues)

```
           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 CR.
           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.
       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.
```

Figure 7-2. COBOL MORTGAGE Program (concluded)

Writing the COBOL Source Program

If you want to try MORTGAGE, 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 COBOL programs in the same place. For example, type

```
) DIR/I ↵
) CREA/DIR  COBOL ↵
) DIR  COBOL ↵
)
```

Now, get into SED or SPEED:

```
XEQ  SED  MORTGAGE.CO ↵
   or
XEQ  SPEED/D  MORTGAGE.CO ↵
```

Using SED or SPEED, type in the program according to Figure 7-2. Don't forget to insert a tab (TAB key or CTRL-I) as needed. (If you chose not to create a COBOL directory, use a filename other than MORTGAGE.CO; for example MORTGAGE.COBOL.CO .)

Compiling the COBOL Program

Now we can compile our mortgage program. Since the initial version probably has some errors, we will ask for a program listing; this listing will help identify the errors.

The command to compile our COBOL program is

```
) COBOL/L  MORTGAGE.CO ↵
```

The /L switch directs the compiler to provide a listing. But you will probably receive the message

```
?FILE DOES NOT EXIST.
)
```

This error occurred because the /L switch specifies the file associated with file @LIST. @LIST is a *generic* file whose name you must set with the LISTFILE command before you can use it. This mechanism, described in Chapter 3 under LISTFILE, is very useful, as you will see.

So let's check the list file setting, then set it to the line printer queuename.

```
) LISTFILE ↵
@LIST
) LISTF  @LPT ↵
)
```

And try the compile again:

```
) COBOL/L  MORTGAGE.CO ↵
```

Given a name of a list file, the compiler can process the program. After a pause, we see

```
5 ERRORS    0 WARNINGS    SEE LISTING.
ERROR
)
```

Checking the line printer, we find two printed sections, each with its own header. The first section is a listing of MORTGAGE.CO with numbered lines; the second is an error report. The error report looks like Figure 7-3:

```
COBOL ERROR LISTING. (W) INDICATES WARNING

LINE    ELEMENT

43        3  MISSING PERIOD FOLLOWING DATA DESCRIPTION ENTRY
92        4  UNDECLARED DATA NAME
92        4  MISSING DATA NAME OR LITERAL FOLLOWING "FROM"
118       4  UNDECLARED DATA NAME
118       4  MISSING RECEIVING FIELD DATA NAME OR LITERAL IN "MOVE" STATEMENT
```

Figure 7-3. COBOL Compiler Error Listing from MORTGAGE

From Figure 7-3, the error situation looks serious. But all 5 errors were caused by only 2 mistakes. The first mistake is easy to find:

On line 43, we wrote SUMMARY-LINE□1. when we meant SUMMARY-LINE1. The space told the compiler that there were two data description entries. The second mistake is trickier. The compiler claims errors on lines 92 and 118, but the syntax of each sentence seems perfect. The UNDECLARED DATA NAME is actually in the data division's file section, in line 13, where we declared OUR-PAYMT-NUM instead of OUT-PAYMENT-NUM. (The compiler /X switch, which produces a cross reference of data names, can help identify this kind of error.)

So, using SED or SPEED, edit MORTGAGE.CO. Correct lines 13 and 43, producing the following changed lines:

```
           02  OUT-PAYMT-NUM  PIC ZZZ9.

01         SUMMARY-LINE1.
```

Leave the editor and type the compile line again, this time without the /L switch.

```
) COBOL   MORTGAGE.CO ⤸
```

Without the /L switch, the compiler writes its messages to the terminal. There are no errors, and the CLI prompt appears when the compilation is complete. Next, build the object file into an executable program file.

Creating the Program File

The CLI command that produces an executable program file from our compiler program is

```
) CBIND   MORTGAGE ⤸
```

As the program file is built, the following message appears on the screen:

LINK REVISION nnnnn ON date AT time
=MORTGAGE.PR CREATED
)

(If you have an older revision of the system, the message will say *AOS BINDER* instead of *LINK*, and include 30 or so module names. Disregard these names and wait for the) prompt.)

Executing the COBOL Program

To execute our COBOL program, or any other program, simply type the XEQ command followed followed by the program name and ⤸.

```
) XEQ   MORTGAGE ⤸
```

MORTGAGE displays

Enter principal:

We then enter a plausible figure and engage in the following dialog with MORTGAGE:

```
Enter principal:     $          50000 ⤸
Interest rate (%):              13.5 ⤸
Years to pay:                   25 ⤸

Type 0 for Summary or 1 for Full Schedule:   0 ⤸
```

MORTGAGE pauses briefly, then says

Type 1 to repeat, 0 to stop.

MORTGAGE dispayed nothing on the terminal because it writes to the list file (@LIST), which we set to @LPT. To see the output, let's get out of MORTGAGE and change the list file:

```
                    0 ⤸
) LISTF ⤸
@LPT
) LISTF  @CONSOLE ⤸
)
```

Having made @CONSOLE (the terminal) the list file, we can see what MORTGAGE is doing. Run it again, and give the same figures, asking for a full schedule:

```
) XEQ   MORTGAGE ⤸

Enter principal:     $          50000 ⤸
Interest rate (%):              13.5 ⤸
Years to pay:                   25 ⤸

Type 0 for Summary or 1 for Full Schedule:   1 ⤸
```

Principal = *$50000.00*
Interest rate = *0.1350*
Loan Life = *25 years*

Monthly payment= *$582.82*

Num	*Interest*	*Prin.Pay*	*Prin.Bal.*	*Interest...*
1	*$562.50*	*$20.32*	*$49979.68*	*$562.50*
2	*$562.27*	*$20.55*	*$49959.13*	*$1124.77*
.	.	.	.	.

It works! Let's 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, execute it again, and give the same figures:

) LISTF MORTGAGE.OUTPUT ⤸
) X MORTGAGE ⤸

Enter principal: *$* 50000 ⤸
Interest rate (%): 13.5 ⤸
Years to pay: 25 ⤸

Type 0 for Summary or 1 for Full Schedule: 1 ⤸

This time, we see nothing because MORTGAGE is writing everything to file MORTGAGE.OUTPUT. After a moment, MORTGAGE says and we reply

Type 1 to repeat, 0 to stop. 0 ⤸
)

MORTGAGE has terminated, closing 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 it, type

) QPRI MORTGAGE.OUTPUT ⤸
QUEUED, SEQ=n, QPRI=n
)

Checking the printer, we find the original summary, then the full schedule, which is about eight pages long. Figure 7-4 shows the beginning of the full schedule. The interest total at the end (which the Figure 7-4 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 is okay, and we need do no 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 MORTGAGE. So, the next time you log on, if you XEQ MORTGAGE and receive a *FILE DOES NOT EXIST* error, simply set the list file to the device you want:

) LIST @CONSOLE ⤸
or
) LIST filename ⤸

You can see the flexibility of the list file, specified in the program on line 8, as ...PRINTER "@LIST." If we hadn't added "@LIST", program output would have been restricted to the line printer. Having said "@LIST", we can select our output file at will from the CLI.

```
Principal =       $50000.00
Interest rate =       0.1350
Loan life =         25 years
Monthly payment =   $582.82

Num       Interest     Prin. Pay  Prin. Bal.     Interest Paid to Date
 1        $562.50       $20.32    $49979.68          $562.50

 2        $562.27       $20.55    $49959.13         $1124.77

 3        $562.04       $20.78    $49938.35         $1686.81

 4        $561.80       $21.02    $49917.33         $2248.61

 5        $561.57       $21.25    $49896.08         $2810.18

 6        $561.33       $21.49    $49874.59         $3371.51

 7        $561.09       $21.73    $49852.86         $3932.60

 8        $560.84       $21.98    $49830.88         $4493.44

 9        $560.59       $22.23    $49808.65         $5054.03

10        $560.34       $22.48    $49786.17         $5614.37

11        $560.09       $22.73    $49763.44         $6174.46

12        $559.84       $22.98    $49740.46         $6734.30

13        $559.58       $23.24    $49717.22         $7293.88

14        $559.32       $23.50    $49693.72         $7853.20

15        $559.05       $23.77    $49669.95         $8412.25

16        $558.78       $24.04    $49645.91         $8971.03

17        $558.51       $24.31    $49621.60         $9529.54
```

Figure 7-4. Beginning of Full Schedule for MORTGAGE (COBOL)

Summary

With the correct listing, you have completed the COBOL chapter -- and have acquired a working knowledge of DG COBOL programming.

What Next?

If you want to try another language, proceed to the appropriate chapter; or, if you want, review earlier material. Or, you can start writing your own programs, using the other pertinent DG COBOL manual.

End of Chapter

Chapter 8
Extended BASIC Programming

This chapter leads you through a sample session in Extended BASIC. Here are the steps you follow to create a BASIC program:

1. Get into BASIC. In some systems, logging on gets you directly into BASIC. If you are logged on into the CLI, type the CLI command:

) XEQ BASIC ⤸

2. Write or edit a series of BASIC program statements. 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 on the run, simply print values on each STOP, then continue the program. Having found the bug(s), go to step 2 and fix the offending statement(s).

6. Write the program to disk with the command LIST "pathname"⤸. Later you can bring it back into memory with the command ENTER pathname⤸.

7. You're done! Type

   BYE ⤸

   to log off the system or get back to the CLI.

About Extended BASIC

A BASIC program is a series of BASIC statements. Each Extended BASIC statement begins with a number between 1 and 9999. BASIC checks the syntax of each line as you type it in. When you type the command RUN⤸, BASIC executes the statements sequentially by number; thus, your program can do useful work.

Extended BASIC variable and array names consist of one letter and -- optionally -- a digit (e.g, A and A1). String variables consist of a letter, optional digit, and $; e.g., S$. You can declare either a string variable or array name with the DIM statement; e.g., DIM R$(30) or DIM B(100). If you omit DIM, the default length of a string variable or array is 10. For comments, you can use either the REM statement or exclamation point (!). With !, you can comment each line directly. For example,

```
10  REM  This is a REM comment. ⤸
20  !  This is a ! comment. ⤸
30  DIM  A(10,10)  ! Dimension array A. ⤸
```

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 command LIST "pathname". Later, you can read it back into memory with the command ENTER "pathname"⤸. From BASIC, you can print the program on the line printer by typing LIST "@LPT"⤸.

To start work on another program, type NEW⤸ and proceed. To sign off BASIC, type BYE⤸.

You can execute a BASIC program only from BASIC; you can't do it from the CLI.

Invoking BASIC

If BASIC comes up on your terminal when you log on (type username and password), skip to the next section.

If the CLI comes up on your terminal when you log on, you'll use the CLI to get into BASIC. Before you can use BASIC, the directory that holds the BASIC files (often directory :UTIL or :BASIC) must be in your search list (SEARCHLIST command, Chapter 3).

Before you start programming in BASIC, we suggest that you create a BASIC 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 ↓
) CREA/DIR  BASIC ↓
) DIR  BASIC ↓
)
```

To get into BASIC, type

```
) XEQ  BASIC ↓
```

BASIC will announce itself:

```
AOS BASIC Revision nnnn basic-date   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 Extended BASIC, try typing in this little program:

```
*  20  print  Test program ↓
ERROR 2 - ILLEGAL STATEMENT SYNTAX
*
```

If you get a numeric error code instead of *ILLEGAL STATEMENT SYNTAX* tell your system manager to build the BASIC ERMES file. (All codes are described in Appendix A of the *Extended BASIC User's Manual.)*

Correct the statement syntax and type in some more statements:

```
* 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 ↓
* list ↓
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

* run ↓
Result is   2.1875
*
```

The uparrow, produced by the SHIFT-6 keys, is the BASIC exponential operator. BASIC evaluated the expression in line 60 via steps 1) A2↑A =16; 2) A1/16=.1875; and 3) A+.1875=2.1875.

Now, write the program to disk, leave BASIC, re-enter BASIC, ENTER the program, and leave BASIC again:

```
* list "testprog" ↓          Write it to disk.
* bye ↓                     Leave BASIC.

username-password           Log on again
  or                          or
) X BASIC ↓                 execute BASIC.

AOS BASIC ...               BASIC banner.
* enter "Testprog" ↓        Bring program into
                             memory.
* bye ↓                     Leave BASIC.
```

Unless you store a program on disk, all work done on it during this BASIC session vanishes when you leave BASIC.

You can use any valid system pathname to list a BASIC program to disk if you have append privileges to the directory(ies) involved; but generally, use a pathname that includes only a filename. If the file named in pathname already exists, BASIC will ask for confirmation; then if you confirm with ↓, BASIC will overwrite the existing file with the new one.

Writing the BASIC Example Program

The BASIC program shown in the flowchart in Figure 8-1 and shown in Figure 8-2 is a variation of the FORTRAN and COBOL programs shown in earlier chapters. The program computes mortgage payments, taxes, and deductions in a general way. It also writes its computations to the terminal and/or the line printer.

The mortgage formulas are those used by banks within the United States; the tax bracket issue is designed for rules set up by the US Internal Revenue Service (IRS). Different mortgage and tax systems are used outside the US: in Canada, Europe, and Asia. So, if you live outside the US, treat the mortgage formulas and tax bracket as an example. (Later, you might want to replace the existing ones with your national ones.)

We assume that you will type in the program from BASIC. (If you have access to the CLI, you can use either the SED or SPEED text editor to type in BASIC progams -- either editor is easier to use than BASIC's. But then you must get into BASIC and enter the program. BASIC will reject all erroneous lines at once, confusing the issue. Later, when you know the system better, you may choose to use SED or SPEED to type in your BASIC programs.)

So, to try the program, get into BASIC. From the CLI, type XEQ BASIC↓; then DIR BASIC↓. Type NEW↓, and then type in the program as shown in Figure 8-2.

```
AOS BASIC ....
*  NEW ↓
*  10  print "This program computes... ↓
   .
```

The NEW↓ command clears your storage area. Before you use the ENTER command, or begin to type in a new program, you should type NEW↓ to clear your storage area and prevent old program lines from intermixing with new program lines. (This can be confusing at the very least.)

As you type in the program, you can examine the lines you've typed by typing LIST. To list a portion of the lines, type LIST number comma number, e.g., LIST 100, 200↓.

Periodically as you type, and when you're done, type LIST "MORTGAGE"↓ (or whatever filename you prefer) to to write the program to disk. You can also get a hardcopy listing by typing LIST "@LPT"↓.

Even if you decide not to type in the program, you should examine Figures 8-1 and 8-2 before proceeding to the next section.

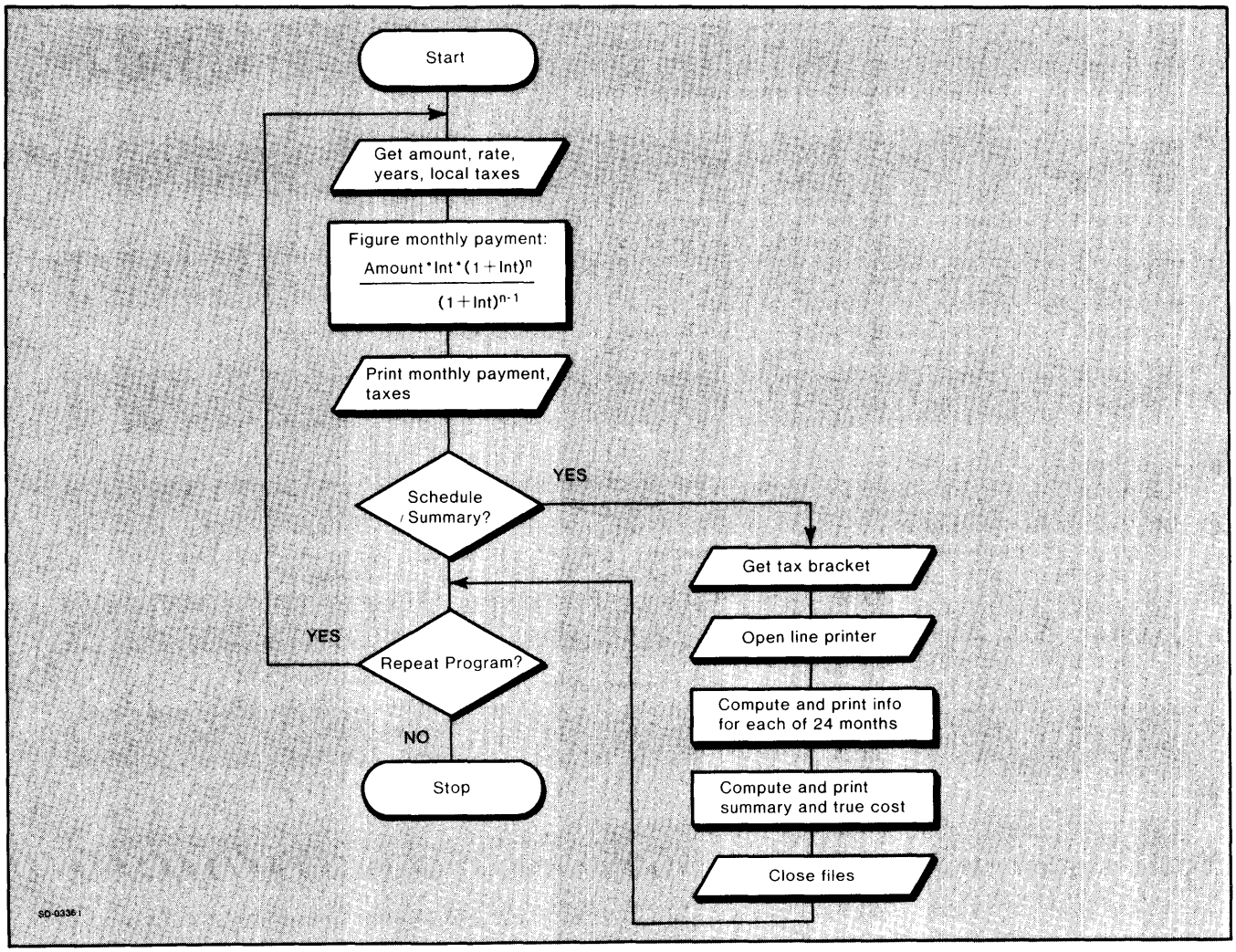


Figure 8-1. MORTGAGE Program Flowchart

```
0010 PRINT "<12> This program computes mortgage payments,"
0020 PRINT "interest, and taxes."
0030 PRINT "Type amount, interest rate, life in years, and"
0040 PRINT "annual property tax. Separate entries with comma,"
0050 PRINT "and with NEW LINE; e.g., 60000, .125, 25, 2000 [NL]"
0060 PRINT
0070 PRINT "   Amount? Rate? Years? Taxes?"
0080 INPUT " ?   ",A,R1,Y,T
0090 LET R=R1/12 ! Make yearly rate R1 monthly rate R.
0100 LET M=12*Y ! Get total number of months M for loan.
0110  !  Compute monthly payment.
0120 LET P=A*R*(1+R)^M/((1+R)^M-1)
0130  !  Define format F$ that rounds numbers to nearest whole cent.
0140 LET F$="--------.##"
0150  !  Print totals and give option for tax subroutine.
0160 PRINT "Monthly payment:      Taxes:          Hideous total:"
0170 PRINT USING F$,P,"  ",T," ",P+T ! Summary info.
0180 PRINT
0190 PRINT "Want a schedule of monthly costs for the first 2 years"
0200 PRINT "and average monthly cost after US tax deductions?"
0210 PRINT "Schedule and summary will go to the line printer."
0220 INPUT "Type capital Y (Yes) for schedule/summary or NEW LINE (No)  ",Q$
0230 IF Q$="Y" THEN GOSUB 1000
0240 PRINT
0250 INPUT "Type Y (Yes) to run program again, or NEW LINE to stop. ",Q$
0260 IF Q$="Y" THEN GOTO 0030
0270 STOP
1000  !   Tax deduction/summary subroutine.
1010 PRINT
1020 INPUT "What is your tax bracket as a fraction (e.g, .24)?  ",B
1030  !   Set up variables for loop. Then open line printer.
1040 LET A1=A ! A1 is amount to be decremented monthly.
1050 LET I1=0 ! I1 is total amount of interest paid to date.
1060 OPEN FILE (0),"@LPA" ! Open on file number 0.
1070 PRINT FILE (0),USING F$,"Rate=";R1*100;"%,  Amount=$";A;", Years=";Y
1080 PRINT FILE (0),USING F$,"Payment=$",P,"  w/tax=$",P+T
1090 PRINT FILE (0),"      Month     Prin.      Int.    Int. total" ! Header.
1100 FOR J=1 TO 24 ! Loop for month = 1 to 24 ---
1110   LET P1=A1*R/((R+1)^M-1) ! Get amount of interest in payment P1.
1120   LET I1=I1+(P-P1) ! Add this month's interest to int total.
1130   LET A1=A1-P1 ! Decrement amount of principal.
1140   LET M=M-1 ! Decrement month M.
1150   PRINT FILE (0),USING F$,J,P1,P-P1,I1 ! Write month's figures to file.
1160 NEXT J
1170 PRINT FILE (0)
1180 LET D=T*12+I1/2 ! Deductions D for 1 yr = T(axes)*12 + (1/2 of 2 yrs int.)
1190 PRINT FILE (0),"Total annual mortgage-related deductions are:"
1200 PRINT FILE (0),USING F$,D
1210 LET D1=D-3400 ! Subtract 0 bracket amount (3400 for marr. filing jointly).
1220 PRINT FILE (0),"After subtracting the $3400 std. (0 bracket)"
1230 PRINT FILE (0),"deduction, the annual mortgage-related deductions are:"
1240 PRINT FILE (0),USING F$,D1
1250  ! Get true mo. cost:  (total mo. pay = P+T) - ((Brkt * D1)/12)
1260 LET C=(P+T)-(B*D1/12)
1270 PRINT FILE (0),"                    ****SUMMARY****"
1280 PRINT FILE (0),"     Life:   Amount:      Rate:  Cash pay:    Brkt:   Cost:"
1290 PRINT FILE (0),USING F$,Y,A;R1*100,P+T,B,C ! Write summary to file.
1300 CLOSE  ! Close open files.
1310 RETURN
```

Figure 8-2. BASIC MORTGAGE Program With Errors

Running the BASIC Program

To run the BASIC program that is currently in memory, type the RUN↓ command:

```
* RUN↓
```

The program executes, announcing itself and asking for MORTGAGE-related information:

This program computes mortgage payments,
interest, and taxes.
Type amount, interest rate, life in years, and
annual property tax. Separate entries with comma,
end with NEW LINE; e.g., 60000, .125, 25, 2000 [NL]

Amount? Rate? Years? Taxes?
?

Respond with some plausible figures:

```
60000, .125,  25,  2000 ↓
```

and the program responds

Monthly Payment	*Taxes:*	*Hideous Total:*
654.21	*2000.00*	*2654.21*

Want a schedule of monthly costs for the first 2 years
and average monthly cost after US tax deductions?
Schedule and summary will go to the line printer.
Type capital Y (Yes) for schedule/summary or NEW
LINE (No)

This seems a little high -- over $2,000 per month. We forgot to convert the yearly tax figure to months. It would be meaningless to proceed, so stop the program:

```
                                ↓
```
Type Y (Yes) to run program again, or
NEW LINE to stop. ↓
STOP AT 270
*

The problem is that we forgot to divide the annual taxes by 12. To fix it, insert a new line of code:

```
*   105 T = T/12   ! Get monthly tax rate.↓
```

and run it again:

```
* RUN↓
```
This program computes ...
... e.g., 60000, .125, 25, 2000 [NL]

Amount? Rate? Years? Taxes?
```
?  60000, .125, 25, 2000↓
```

Monthly Payment	*Taxes:*	*Hideous Total:*
654.21	*166.67*	*820.88*

Want a schedule of monthly costs...
and average monthly cost after US tax deductions?
Schedule and summary will go to the line printer.
...capital Y (Yes) for schedule/summary or NEW LINE
(No)

These figures are more reasonable, so we can proceed with the program:

```
                        Y↓
```
What is your tax bracket as a fraction (e.g, .24)?

The tax bracket issue is explained later in this chapter. For now, try .24, a typical bracket:

```
                        .24↓
```

I/O ERROR AT 1060 - WRITE ACCESS DENIED
*

A runtime error! Check the offending line:

```
* list  1060 ↓
```
1060 OPEN FILE(0),"@LPA"   ! Open on ...
*

The problem is that we 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 it, 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 1060 that references the printer queuename:

```
* 1060 Open file(0),"@LPT"   ! Open on file number 0.↓
*
```

Run it again, giving the same figures (60000, .125, 25, and 2000) and tax bracket (.24).

This time there is no runtime error, but there is a slight delay as the program writes the schedule to the printer. Then it says:

Type Y (Yes) to run program again, or
NEW LINE to stop.

Because the program closed the printer file, the output should appear immediately on the printer. Checking the printer, we find the 24-month schedule with tax computation. This schedule is shown in Figure 8-3.

A header precedes the printed schedule. The header has our username, pathname to the temporary printer file, time, and date.

You've fixed the program, so you can stop it and write it to disk using its original name. BASIC asks for verification of the overwrite and you provide this by typing ⤶:

```
                    ⤶
STOP AT 270
* LIST  "MORTGAGE" ⤶
TYPE NL TO DELETE OLD:    ⤶
*
```

You can also print the program from BASIC by typing LIST "@LPT".

To leave BASIC and return to the CLI (or to log off, if you logged on into BASIC), type

```
* BYE ⤶
```

Summary

You have now completed the Extended BASIC program exercise -- and have acquired a sound basis for DG BASIC programming. (For tax bracket information, read the next section.)

What Next?

If you want to try another language, proceed to the appropriate chapter; or, if you want, review earlier material. Or, you can start writing your own programs, using the other pertinent DG BASIC manuals.

```
Rate=          12.50%,  Amount   60000.00, Years=        25.00
Payment=$     654.21  w/tax=$     820.88
      Month     Prin.      Int.     Int. total
       1.00     29.21    625.00      625.00
       2.00     29.52    624.70     1249.70
       3.00     29.82    624.39     1874.08
       4.00     30.13    624.08     2498.16
       5.00     30.45    623.76     3121.93
       6.00     30.77    623.45     3745.37
       7.00     31.09    623.13     4368.50
       8.00     31.41    622.80     4991.30
       9.00     31.74    622.47     5613.77
      10.00     32.07    622.14     6235.92
      11.00     32.40    621.81     6857.73
      12.00     32.74    621.47     7479.20
      13.00     33.08    621.13     8100.33
      14.00     33.43    620.79     8721.12
      15.00     33.77    620.44     9341.56
      16.00     34.13    620.09     9961.65
      17.00     34.48    619.73    10581.38
      18.00     34.84    619.37    11200.75
      19.00     35.20    619.01    11819.76
      20.00     35.57    618.64    12438.40
      21.00     35.94    618.27    13056.68
      22.00     36.31    617.90    13674.58
      23.00     36.69    617.52    14292.09
      24.00     37.07    617.14    14909.23

Total annual mortgage-related deductions are:
   9454.62
After subtracting the $3400 std. (0 bracket)
deduction, the annual mortgage-related deductions are:
   6054.62
                  ****SUMMARY****
    Life:    Amount:       Rate:  Cash pay:     Brkt:    Cost:
    25.00  60000.00      12.50     820.88       .24     699.79
```

Figure 8-3. Summary/Tax Schedule from MORTGAGE Program (BASIC)

Itemized Deductions and Tax Bracket

For simplicity, the sample BASIC program assumes you are married and, when you acquire this mortgage, you will start itemizing deductions instead of taking the standard deduction. When you itemize, the IRS allows you to deduct only the itemized amount over the standard deduction ($3400 if married and filing jointly, $2300 if single). The standard deduction is already figured into the tax tables. This is why line 1210 of the program subtracts the standard deduction from the mortgage-related deductions. If you are, in fact, moving to itemized deductions, you can deduct much more than mortgage-related expenses (e.g., medical expenses, casualty losses), but the program doesn't deal with these. It figures the true *Cost* amount as if you were deducting *only* the mortgage-related amounts.

Thus, if the mortgage moves you from the standard deduction to itemized deductions, your real cost per month will be less than the true *Cost* figure. If this is true for you, you can modify the program to compute the TRUE figure more accurately. The critical figure is the 3400 in line 1210. Your program statements should get all mortgage-unrelated deductions (medical, contributions, casualty losses, etc.) and put them in a variable, let's say Q. Then it should *add* Q to D in line 1210; e.g., LET D1 = D+Q-3400.

In any case, if you are single, change the 3400 in line 1210 to 2300; e.g., LET D1 = D-2300.

Tax Bracket

It's easy to find your tax bracket if you don't know it. Take the figure you used in the tax tables to find your tax, and subtract 3400 (married filing jointly) or 2300 (single); this is your gross income less deductions. Multiply the number of exemptions you used to find your tax by 1000, and subtract the product from the original figure. This is roughly your net ("taxable") income.

Now look at the Tax Rate Schedules (not the Tax Tables) in the current US Form 1040 instruction book. In Schedule X or Y (single or married), 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.

For example, assume that the figure you took to the tax tables was $23,000 and that you have three exemptions. 23,000 minus the standard (marital) deduction of 3,400 is 19,600. 19,600 minus 3*1000 is 16,600. In Tax Rate Schedule Y, you find that 16,600 falls between $16,000 and $20,200. Reading to the right on the same line, you find $2,265+24%. Thus your tax bracket is 24% (.24) because the government took 24 cents from the last taxable dollar. Thus you save 24 cents for each dollar the new mortgage lets you deduct. (This procedure yields only an approximation -- because the mortgage deductions may well drop you into a lower bracket, thus save you less in deductions -- but it is a fairly close approximation.)

End of Chapter

069-000018

Chapter 9
Assembly Language Programming

This book can't teach you all about programming in DG assembly language, but this chapter will introduce some of the basics. We assume that you have some familiarity with the mechanism of assembly language and with the ECLIPSE™ computer instruction set (detailed in the appropriate series 14 hardware *Programmer's Reference Manual*).

But even if you are not familiar with these, you will profit by reading this chapter, for it will give you the fundamental concepts before you study them in greater detail. Also, you'll need some of the basics described in this chapter to fully understand the next chapter where we write, assemble, execute, and debug our own assembly language program.

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 MASM, the macroassembler.

The program development sequence looks like this:

1. Create or edit an assembly language source file (sometimes called a *module)* with SED or SPEED:

) XEQ SED pathname ⤶
   or
) XEQ SPEED/D pathname ⤶

   Via your chosen editor, type in the assembly language statements that make up the program.

2. Assemble the source with the CLI command

) XEQ MASM/L=@LPT pathname ⤶

3. If there are assembly errors, return to step 1 and fix the offending statement(s). If there are no errors, go to step 4.

4. Link the object file to produce an executable program:

) XEQ LINK pathname ⤶

5. Execute the program with the CLI command:

) XEQ pathname *[arguments]* ⤶

6. If the program runs as you want it to, go to step 9.

7. Try to identify the problem(s) using runtime error messages or erroneous output. If you succeed, go to step 1 and fix the problem(s).

8. Debug the program with the command:

) DEB pathname *[arguments]* ⤶

   The Debugger is a utility that allows you to stop the program, examine memory locations, and make temporary changes. Having used the Debugger to find the problem(s), go to step 1 and fix them.

9. You're done!

You will probably have to repeat the process of assembly, Linking and execution/debugging several times before you produce a program that works as you want it to.

Introduction to the Macroassembler (MASM)

MASM, the Macroassembler, is a system utility program that translates one or more source program files (called modules) into instructions the machine can understand.

Each module employs symbolic instruction codes (like LDA 0,FOO for "load the contents of location FOO into accumulator 0") and operating system calls (like ?READ for "read a record"). Your modules will also use macroassembler pseudo-operatives (pseudo-ops), which direct the assembly process but do not result in any final program instructions themselves.

MASM is a two-pass assembler (i.e., it examines the modules twice), and at the end of the second pass, it produces one or more of the following:

- an assembly listing of the module(s)
- an error code listing
- a binary object module (OB)

The assembly listing shows you your original source module(s) and additional information like the octal codes of your instructions and data, the locations of these items in the executable program, and other miscellaneous information.

Assembly errors are analagous to FORTRAN or COBOL compilation errors. They usually indicate syntactical errors, not errors of logic in your program. By default, error messages are displayed on your screen. The listing will also contain one or more error codes beside each offending source line.

After you have produced an object module with no assembly errors, you then Link it (perhaps along with other modules) to produce an executable program. You then run the program.

Incidentally, the Macroassembler derives its name from the fact that it lets you define and use instruction sequences called macros. After defining a sequence as a macro, you can use simply the macro name instead of all the instructions it contains. But you don't need macros to program, so we do not describe them in this book.

Understanding Program Listings

Each line of source code that you write will have one or more of the following elements:

- labels (names of locations)
- instructions, system calls, or pseudo-ops
- arguments (separated by spaces or commas)
- comments (preceded by a semicolon)

The Macroassembler accepts free-form input, hence you need not place any of these elements in a specific line position. The only requirement for source code is that, in any line where they are present, the order of these elements must be *label, instruction, arguments* and *comment*.

Each *label* ends with a colon. For our purposes, MASM considers only the first five characters of each label to be significant (it does not, for example, distinguish the labels BEGIN and BEGINNING. (There is a switch that instructs it to distinguish eight chararacters.)

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 9-1 shows you a page from an assembly listing. Typically, such a listing is obtained with the following command:

```
) XEQ  MASM/L=@LPT  pathname ↵
```

where pathname is an assembly language source module.

Notice, in Figure 9-1, that there are three columns of numbers; to the right of these you can see the source code that was input to MASM. The source code starts in column 17 of the listing and extends to column 80. The enlarged view of the listing shows three source lines that use each of the source code elements:

```
label          instruction/argument    comment

               JMP      ERTN           ; ERROR
CONT:          LDA      2, ?OAC2, 3
               LDA      2, 0, 2        ; GET RE
```

The label is at the left, the instructions next, argument (third line) next, and the comments are on the right. We aligned labels, instructions/arguments, and comments to make the listing easier to read, but we need not have done so; MASM requires only that you follow the sequence of label (if any), instruction, arguments (if any), and comments (if any) within a line. (You can see, in Figure 9-1, that the argument to the JMP instruction (ERTN) does not line up with the arguments to the two LDA instructions which follow it.)

Figure 9-1 also shows you what the columns of numbers mean. Columns 1 and 2 contain the line numbers for information displayed on each page. The assembler starts a new listing page after the 60th line unless you inserted a form feed character before this line. At the top of each page you will see a four-digit number. This is the page number. MASM uses page and line numbers in the alphabetical cross-reference it produces with each program listing (shown in Figure 9-2). Now return to Figure 9-1. If any source line has one or more assembly errors, columns 1 to 3 will contain from 1-3 alphabetic error codes.

The next row of numbers is in octal and extends from columns 4 through 8. This sequence shows where each line of code will reside in the final program. These numbers may indicate absolute values (e.g., memory address 00017) or they may indicate only a relative position (the 3rd location from the beginning of this series).

Columns 10 through 15 show the octal assembled value that will reside in this location, if this value is known when the program is assembled. If MASM doesn't know this value, perhaps because it references another module, the listing shows the contents as 00000.

The two remaining columns, 9 and 16, contain relocation flags. The symbols ' and " indicate the types of relocatability. Relocation is described in greater detail in the *Link User's Manual*.

Figure 9-2 shows you a sample page from a cross-reference listing.

The cross reference lists the page and line number where each user symbol (such as a label) is defined, and it lists the page and line number of each reference to this symbol. The column labeled "type of symbol" indicates which symbols were externally defined or entered. We will explain symbol types when we discuss pseudo-ops.

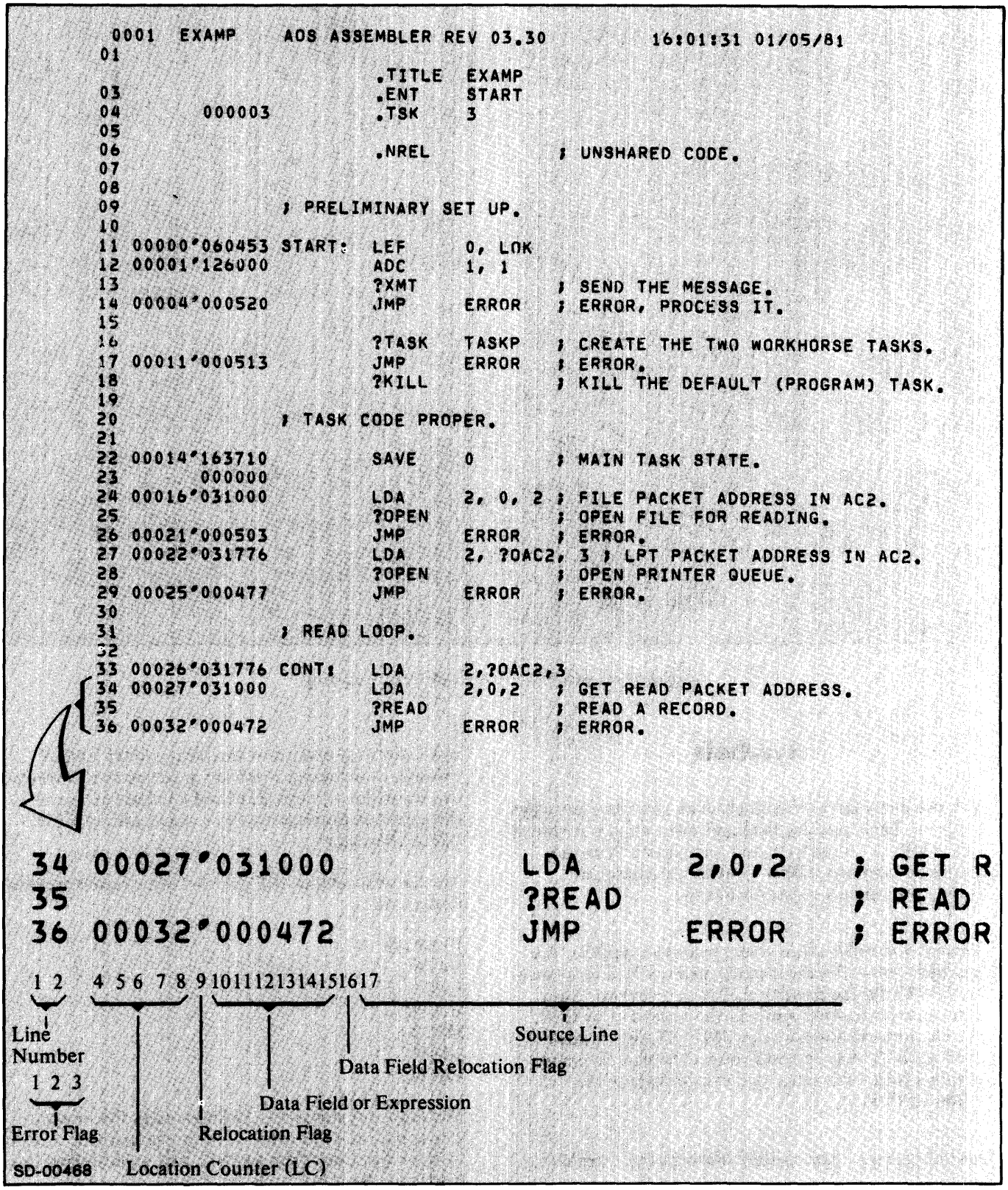


Figure 9-1. Assembled Program Listing

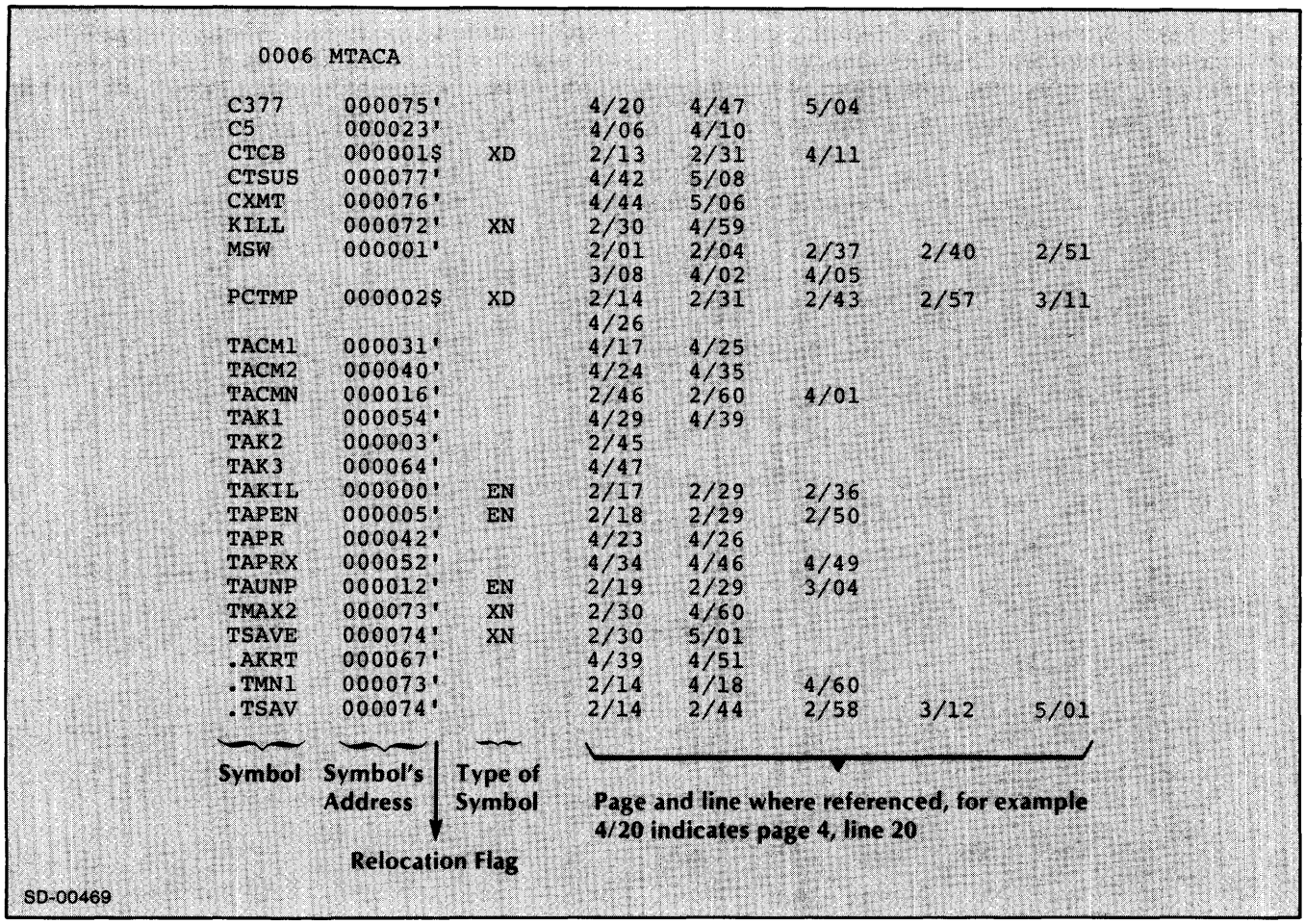


Figure 9-2. Assembler Cross-Reference Listing

Symbols

You use many kinds of symbols in an assembly language program. Each machine language instruction is a symbol (like JMP for "jump to"), and some instructions use predefined symbols. Other symbols are assembler pseudo-ops; these are described later.

Finally, you will want to create your own symbols. For example, you will want to name memory locations (such as START for the beginning of your program). Then you can refer to them symbolically instead of by their precise numeric address (e.g., JMP START instead of JMP 01762). 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 the first-character can be any letter from A through Z, including a period (.); *succeeding characters* can include A through Z, period (.), and numbers 0 through 9. A break is usually a space, tab, or NEW LINE character.

The Symbol column of Figure 9-2 shows some symbols. Others are

□.DATA
DATA
FILE1
BT000
MESS1
.MSG1
FATAL

While your symbols may be longer than five or eight characters, MASM considers only the first five or eight to be unique (labels are symbols, and we described this restriction for labels earlier). In this book, we use symbols of five characters or less.

You may not use a period (.) as a single-character symbol. MASM reserves this symbol to mean the value of the current location in a program. For example, JMP .-1 means "Jump 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, do not use question mark (?) as the first character of any symbol. The operating system defines a set of symbols (including system calls) for your use, and it uses ? as the first character of each symbol. All symbols that begin with ? in Figure 9-1 are symbols defined for the operating system.

If you follow these rules you may still receive an occasional assembly error indicating a bad symbol. This may happen because there are symbols (like 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 at its beginning or end.

Argument Operators

From time to time, you may want to perform some operations on symbols. Here are some operators that you can use with symbols, integers, and the current location counter(.):

	Operator	Meaning
Bit	B	Bit alignment; e.g, 1B0+1B7 sets bits 0 and 7 to 1.
Arithmetic	+	Addition (e.g., 2+3) or unary plus, (e.g., 2-(+3)).
	-	Subtraction (e.g., 5-2) or unary minus, (e.g., 5-(-2)).
	*	Multiplication.
	/	Division.

Many *system* symbols use the B operator, and you in turn use the symbols; for example, symbols ?RTDS+?OFIO use B to set two bits in meaningful ways.

Arithmetic operators are useful with address offsets: you could write JMP .+2 to jump 2 words from the current location. If your program reserves a series of addresses whose first word is address TABLE, indexes off 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 (as detailed in the next chapter).

In terms of precedence, if you use more than one operator, MASM evaluates the B operator first, then operators in parentheses left to right, then operators outside parentheses left to right. So, to 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), while 10. equals decimal 10 (octal 12).

Accumulators (Registers)

All ECLIPSE computers have four 16-bit registers called accumulators. We call these AC0, AC1, AC2, and AC3; you reference them as 0, 1, 2, or 3 within instructions. Accumulators are often used to reference addresses and are *always* used for address manipulation and comparison. All four ACs work pretty much the same way, except that AC2 and AC3 provide address indexing; e.g.,

```
LDA    0, -10, 2
```

means "load AC0 with (the value in AC2, -10)".

Instruction Types

Most ECLIPSE machine instructions assemble into one 16-bit storage word. There are three types of instruction:

- *Memory Reference Instructions (MRIs).* MRIs concern memory locations or their contents. They permit one of four different kinds of indexing into these locations. For some MRIs, the address referenced must be within 128 words; JMP, LDA, and STA are such instructions. Variations of these instructions can reference any address in the program; they assemble into two 16-bit words, and examples are EJMP, ELDA, and ESTA. Some basic MRI instructions and their *simplest* formats are shown in Table 9-1.

- *Arithmetic-Logical Instructions* (ALCs). ALCs can add and subtract values, shift bits, swap bytes, use the overflow (Carry), and redirect program execution. They can also AND values to mask portions of words. All ALC instructions require a source accumulator and a destination accumulator to receive the result of the arithmetic-logical operation. Examples of ALC instructions are MOV 0, 1 (copy the contents of AC0 to AC1) and SUB # 0,1 SZR (subtract the contents of AC0 from AC1, don't load AC1 with the result, skip the next instruction if the result is 0). Some common ALC instructions are shown in Table 9-2.

- *Input-Output (I/O) Instructions.* 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, which are outside the scope of this book.

Table 9-1. Common MRI Instructions

Instruction and Format	Explanation	Example
JMP addr EJMP addr	Jump to address addr, which can be a symbolic address or expression. JMP assembles into one 16-bit word and its addr must be within 127 words forward or backward; EJMP assembles into two 16-bit words and its addr can be anywhere in the program. This definition of addr applies to all instructions.	JMP ERR JMP LOOP+6
JSR addr EJSR addr	Jump to addr, save return address (.+1) in AC3. Normally used to enter a subroutine. The subroutine can save AC3 with a store instruction; then, when it wants to return, load AC3 with the stored address, then issue JMP 0,3 to return.	JSR SUBR EJSR SUB2
LDA ac addr ELDA ac addr	Load accumulator (ac) with the contents of addr.	LDA 2, FNAME ELDA 2, FNAME
STA ac addr ESTA ac addr	Store the contents of ac in addr.	STA 2, TSAVE ESTA 2, TSAVE
DSZ addr	Decrement the contents of addr by 1; skip the next word if the contents of addr have reached 0.	DSZ COUNT

Table 9-2. Common ALC Instructions

Instruction and Format	Explanation	Example
ADD *[opt]* acs acd *[s]*	Add the contents of accumulator acs to contents of acd; place result in acd. *opt* is one of the carry, shift, or no-load options; *s* is an optional skip directive. These definitions of *opt* and *s* apply to all ALC instructions.	ADD 0,2 ADDL 0,2,SNR
SUB *[opt]* acs acd *[s]*	Subtract the contents of accumulator acs from contents of acd; place result in acd.	SUB 0,2 SUBC 0,2,SNR
MOV *[opt]* acs acd *[s]*	Copy the contents of accumulator acs to acd. Often used simply to check or manipulate an ac.	MOV 0,2 MOVL 0,0,SNC
AND *[opt]* acs acd *[s]*	Logically AND the contents of accumulator acs with contents of acd; place result in acd.	AND 2,3

Special Instruction Symbols

You can use certain symbols to modify the operation of certain arithmetic/logical and memory reference instructions as shown under the *opt* category in Table 9-2.

You can append *one* of the following one-character symbols as a suffix to any arithmetic/logical (ALC) instruction to produce the following results:

Opt	Meaning	Result
C	Complement carry	Change the carry bit after the ALC operation.
L	Left shift	Combine the carry bit with the ac and shift the 17-bit result one bit left, *after* the ALC operation.
O	Set carry	Set the carry bit to 1 before the ALC operation.
R	Right shift	Combine the carry bit with the ac and shift the 17-bit result one bit right, *after* the ALC operation.
S	Swap	Swap the left and right bytes of the ac *after* the ALC operation.
Z	clear carry	Clear the carry bit to 0 *before* the ALC operation.

Often, to redirect control, you may want to skip the next instruction; for example, if some condition is true or not true. You can use the following 3-character skip symbols with ALC instructions to skip the next *single word* location in your program (don't specify a skip before an instruction that assembles as two words; e.g., EJMP). You must always place these symbols at the end of the ALC argument list.

Symbol	Result
SKP	Unconditionally skip the next instruction.
SZC	Skip the next instruction if the carry bit equals 0.
SNC	Skip the next instruction if the carry bit equals 1.
SZR	Skip the next instruction if the result equals 0.
SNR	Skip the next instruction if the result doesn't equal 0.
SBN	Skip the next instruction if *both* the carry bit and the result are nonzero.
SEZ	Skip the next instruction if *either* the result or carry bit equals 0.

You will see uses of these special instruction symbols in the example program in Chapter 10.

Special Characters

There are two special operations you can specify when using some machine language instructions: indirect addressing, and no-load of the result of arithmetic/logical operations.

In a source program line containing a MRI instruction (like LDA or JSR), you can use the commercial at sign (@). An @ anywhere in an MRI argument sets bit 5 in that instruction, the indirect addressing bit. For example:

```
024060    LDA  1, 60
026060    LDA  1, @60
```

The first instruction loads AC1 with the contents of memory location 60; the second loads AC1 with the word whose *address* is in location 60.

A number sign (#) may appear anywhere in an ALC (arithmetic/logical) instruction. A # specifies the *no-load* option by setting bit 12 of the instruction to 1. The no-load option directs the system *not to load* the destination accumulator, but otherwise to perform the operation as usual. This is useful when you want to test for equality, zero or other values without changing the value in the accumulator.

```
133102    ADDL  1, 2, SZC
133112    ADDL# 1, 2, SZC
```

The first instruction adds the contents of AC1 to AC2, shifts the result left one bit, places the result in the carry bit and AC2, and skips the next instruction if the carry bit equals 0; the second instruction does the same thing, but does not load AC2 with the result. in the first instruction, AC2 is loaded with the result; in the second, it is unchanged.

For the no-load option to work, you must also specify one of the skip symbols (SZR, etc.).

The following examples show how you can use special instruction symbols and special characters to modify the ADD instruction.

Instruction	Meaning
ADD 1, 2	Add the contents of accumulator AC1 to AC2.
ADDL 1, 2	Add the contents of AC1 and AC2, shift the result one bit to the left, placing the carry in bit 15, and leave the result in AC2 and carry.
ADDL 1, 2, SZC	Add the contents of AC1 and AC2, shift the result one bit to the left, leave the result in AC2 and carry, and skip if this operation yields a zero carry.
ADDL# 1, 2, SZC	Add the contents of AC1 and AC2, shift the result one bit to the left, and skip if this operation yields a zero carry. Do not alter the original contents of AC2.

Pseudo-ops

A *pseudo-op* instruction directs the operation of the assembler. It is called a "pseudo-operative" because your program never executes it. The pseudo-ops that you need to start assembly language programming are

Pseudo-op	Function
.BLK	Reserve a block (series) of 16-bit words.
.END	End a module and (optionally) name a starting address.
.ENT	Declare an entry point or symbol to be available for other modules' use.
.EXTN	Declare an external entry point or 16-bit symbol (point or symbol defined in some other module).
.LOC	Commence assembly of code and data at a specified location.
.NREL	Assemble the following code and data for execution in unshared memory (Normal RELocatable). (.NREL can also specify shared memory, but this is outside the scope of this book.)
.TITL or .TITLE	Assign a title to a module.
.TXT	Create an ASCII text string.
.ZREL	Assemble the following code for execution in lower page zero (Zero RELocatable).

Often you will want to refer to a symbol or call a subroutine that is defined outside the module you are writing. To do this, you can name the symbol or entry point in an .EXTN pseudo-op.

Each argument that you name in an .EXTN pseudo-op must be made available with an .ENT pseudo-op in another module that you will Link along with this module.

The .TXT pseudo-op assembles a text string into its equivalent ASCII codes (two per word), terminated by a null byte. You may 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<null> in the third and final word.

.TITL (or .TITLE) assigns a title to a module. This has no direct relation to the module's filename. The title appears on the top line of each page of the module's assembly listing.

.ZREL starts assembly storage from the first available ZREL location (location 50_8 by default). ZREL (also called lower page zero) storage extends through from 50_8 to 377_8 and is directly addressable by all instructions. .NREL starts assembly storage at the beginning of unshared NREL memory. The start of NREL storage varies, as determined by Link.

Now that we have described certain Macroassembler pseudo-ops we can explain the third column of each assembly listing cross reference (see Figure 9-2). This column contains two-letter codes describing the types of symbols which have been the arguments of certain pseudo-ops:

Code	Meaning
EN	Entry (.ENT pseudo-op)
XN	External normal (.EXTN pseudo-op)

XD and other codes, outside the scope of this discussion, refer to pseudo-ops that we have not described.

.BLK
Allocate a block of storage

Format:

.BLK expression

Description:

This pseudo-op allocates a block of memory storage. expression is the number of words you want reserved; the current location counter is incremented by expression.

Example:

```
TABLE:          .BLK 60.        ; Block for TABLE.
```

This example reserves a block of 60_{10} memory words; the first word in the series has the symbolic name TABLE.

```
PAKET:          .BLK    ?IBLT ; Block for packet.
```

This example reserves a block of ?IBLT memory words; the first word has the name PAKET. You will see this use of .BLK in the declaration of system parameter packets in the next chapter.

.END
Indicate 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 .END, indicating the address that will receive control when the program is executed.

Example:

```
START:          ?OPEN  TERM   ; Open terminal.
                .
                .
                .END  START
```

In this example, the .END pseudo-op terminates the module and defines as the address which will receive control when you run the program.

.ENT
Define a module entry

Format:

.ENT symbol *[,symbol] [...]*

Description:

This pseudo-op declares user symbols defined within the module to be available for referencing 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 can use the .EXTN pseudo-op reference **symbols** defined by the .ENT pseudo-op.

In the example, PROGA is the main program. It can call two external routines in module TRIG: SINE and COS. The pointers .SINE:SINE and .COS:COS enable the JSR instructions to can reach the address of SINE and COS no matter where they are. Without a pointer, a JSR (but not an EJSR) can reach forward or backward only 127 locations. 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 ↵
.
```

Example:

```
          .TITL   PROGA    ; PROGA is main module.
          .EXTN   SINE, COS  ; SINE, COS are in module TRIG.
          .NREL
START:    .                ; PROGA initializes things here and
          .                ; does its own processing until it needs
          .                ; routine SINE or COSINE.
          JSR     @ .SINE  ; JSR to SINE in other module.
          .                ; Return here and continue.
          JSR     @ .COS   ; JSR to COS in other module.
          .                ; Return here and continue.
.SINE:    SINE             ; Pointer to SINE address.
.COS:     COS              ; Pointer to COS address.
          .
          .END    START
```

```
          .TITL   TRIG          ; TRIG does trig.
          .ENT    SINE, COS     ; Identify entries.
          .NREL

SINE:     ESTA    3, SAVE3      ; Save return address.
          .                     ; SINE processing routine is here,
          .                     ;  uses AC3.
          ELDA    3, SAVE3      ; Done with SINE code, restore return address.
          JMP     0, 3          ; Return to calling program module.
COS:      ESTA    3, SAVE3      ; COS routine also saves return address.
          .                     ; Do COSine processing.
          ELDA    3, SAVE3      ; Restore return address.
          JMP     0, 3          ; Return to calling program module.
SAVE3:    0
          .END
```

.EXTN
Declare an external reference

Format:

.EXTN symbol *[,symbol] [...]*

Description:

This pseudo-op declares that one or more symbols referenced by this module are defined (with the .ENT command) in other modules.

Example:

See the .ENT example.

.LOC
Set the location counter

Format:

.LOC expression

Description:

This pseudo-op sets the location counter value to expression.

MASM has an elaborate set of rules describing what combinations of relocation types you can mix and what operations you can use in the expression.

Generally, for .NREL (normal relocatable code), you should use only symbols -- not constants -- within the 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 the correct constants (as you will see in the next chapter).

For .ZREL (page zero code), you can use constant expressions if you wish, but, preferably, will use symbols.

Example:

The following example is an *assembled* listing.

```
                        .ZREL ; Page zero.
00000-000572    C:      378.
                        .
                        .NREL ; Normal unshared.
00000'030000-   START:  LDA 2, C
                        .
                        .
                        ?OPEN TERM ; Sys. call.
                        .
                        .
00151'000014    TERM:   .BLK  ?IBLT ; Packet.
      000152'           .LOC  TERM+?ISTI ; Symbolic addr.
00152'040030            ?ICRF+?OFIO      ; Sys. symbols.
      000154'           .LOC  TERM+?IBAD ; Symbolic addr.
00154'000352"           BUFF*2           ; User symbol.
      000165'           .LOC TERM+?IBLT ; End of packet.
00165'044151    BUFF:   .
                        .
```

In this example, Link will place C in the first available word of ZREL memory. 00000 in the address column of the listing indicates the first ZREL address (usually location 50_8), not absolute location 00000. Likewise, 00000 in the NREL section indicates the first available NREL location.

Note, within the system packet, that the location counter moves from 00152 to 00154 and, finally, at the end of the packet to 00165. This shows how the system-defined symbols allot needed space -- reserving locations that the system may need, even though the user doesn't need or care about them.

.NREL
Set the location counter to unshared code

Format:

.NREL

Description:

The .NREL pseudo-op sets the location counter to the first word of unshared memory. It is normally required in each module. .NREL has other uses, but these are outside the scope of this book.

Example:

```
        .TITL       SPROG
        .EXTN       MAIN, ERR
        .ENT        SUBR, SUBR1

        .NREL       ; Normal relocatable
                    ; code.
INIT:   .
        .
        .END
```

Also see the .LOC and .ENT examples.

.TITL or .TITLE
Entitle an object module

Format:

```
 .TITL    symbol
or
 .TITLE   symbol
```

Description:

This pseudo-op gives an object module a title, 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 relation to the module's filename, but some people use the same names for clarity.

Example:

```
        .TITL EXAMPLE
        .NREL
INIT:   .
        .
        .END   INIT
```

This example assigns the title EXAMP to this module and prints EXAMP at the top of each listing page. (MASM, which considers all symbols to be unique within the first five characters only, will discard the LE of EXAMPLE; the listings will start with EXAMP.)

.TXT
Create 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. You can put nonprinting characters in the string by enclosing their ASCII values in angle brackets. You can also use lowercase letters and have them output in lowercase.

If you insert a NEW LINE character (<12>) in the string, the system may treat it as a record delimiter, and not output the rest of the line. To make sure the system outputs the whole line, simply add 200_8 to NEW LINE's ASCII value; for example:

```
.TXT "This<212>
  is a multiline<212>
  text string.<12>"
```

Examples:

```
.TXT   "Abcde"
```

This example creates the ASCII string consisting of three words. The first contains the letters **Ab**, and the second contains the letters **cd**. The last word contains an **e** in the left byte and a terminating null in the right byte.

```
.TXT   "Ab<011>cde"
```

This example creates the ASCII string consisting of the letters **Ab** and **cde**, separated by a horizontal tab (ASCII code 011).

.ZREL
Set 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:

```
                                .TITL   PROGC

                                .ZREL   ; Put pointer in ZREL.
00000-037522' ER:               ERROR   ; Point to ERROR addr.

                                .NREL
                      START:    ?OPEN   CON ; Open terminal.
00004'002000-                   JMP     @ER
00005'037200                    .
                                .
                                ?WRITE  CON ; Message.
37211'002000-                   JMP     @ER
                                .
37522'030404 ERROR:             .           ; Error handler.
                                .
37525'000014 CON:               .BLK ?IBLI
                                .
                                .
```

This example shows how to place a pointer in page zero, and through, the pointer, reference any address in your program. Through the pointer, the ?OPEN in NREL line 00004 can access the ERROR routine, some 37000_8 locations away. A simple JMP ERROR would not have worked here, producing an assembler A (addressing) error.

Wherever you can use EJMP, EJSR, ELDA, or ESTA instructions, you don't need a pointer. But, because system calls require a *one-word* error return location, you cannot use the E-instructions to handle system call errors. Thus you may want to use ZREL pointers often to handle error returns from system calls.

What Next?

After reading this chapter, you have the primary background needed for the next chapter: Programming AOS. Proceed.

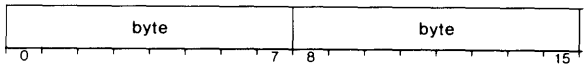
End of Chapter

Chapter 10
Writing AOS Assembly Language Programs

In this chapter we talk about words and bytes and introduce some AOS operating system calls, with examples. Next, we will write an assembly-language program, assemble it, correct the assembly errors, Link it, and execute it. Finally we will debug the program.

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 byte string consists of a sequence of bytes, packed left to right in a series of one or more words.



You can form a *pointer* to a byte (a byte address or *byte pointer*) by multiplying the starting address of the text string by 2. For example:

```
        TEXT*2      ; Byte pointer is TEXT*2.
        .
        .
TEXT:   .TXT  "This is a text string."
```

To *load* or *store* a byte pointer, there are special instructions not described here. We will use the following mechanism to do it:

```
        LDA  ac, TXP     ; TXP is word
                         ; with byte address.
        .
TXP:    TEXT*2       ; Points to 1st byte.
TEXT:   .TXT   "This is a text string."
```

System Call Format

Each system call is a DG-supplied assembly language macro. 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 to from 2 to 6 locations. So your program listings will show 2 to 6 locations for each system call.

The exception (error) return will receive control on either an error condition, or an exceptional condition like end-of-file. This must be a *one-word* instruction that tells the system what to do if an error occurs (for example, JMP ERROR where ERROR starts an error-handling routine). Because the exception instruction must be one word long, you can't use long-range memory reference instructions (MRI) like EJMP for it -- because these assemble into *two* words. You must always reserve an exception return word, even if you envision no possible exception condition.

If the call executes normally, control goes to the normal return -- and the program proceeds.

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 by typing

```
) MES  code-number ↵
```

to the CLI. Some common error codes are described later in this chapter.

System calls that involve I/O take, as an argument, the address of a *parameter packet;* e.g.,

```
?call-name      packet-address
exception (error) return
normal return
```

When AOS executes a call, that involves I/O, it places the parameter packet address in AC2 before doing 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 location 41_8 (the frame pointer, FP); thus AC3, too, is overwritten. So, unless an error occurs, both AC0 and AC1 will remain unchanged throughout a system call.

Some operating system calls are executed in user address space (modules are automatically taken from a library named URT.LB for this), and other calls are executed within the system itself. You must not use memory locations $0\text{-}15_8$ or 17_8: the system uses these locations for call processing.

Operating System Calls

We describe here only the most important features of these calls. Later, you may want to use other features. For many applications, however, the simple versions will suffice.

The calls we describe here are

?OPEN	Open a file or device.
?READ	Read a record (e.g., a line of text).
?WRITE	Write a record (e.g., a line of text).
?GTMES	Read from the CLI command line.
?RETURN	Stop the program and return to the CLI.

System call ?OPEN opens a file -- or device like the terminal -- for access. ?OPEN can also create a file. Calls ?READ and ?WRITE read information from the file and write it to the file.

The ?GTMES call lets the program read arguments from the CLI command line that executed the program; e.g., to read FOO from XEQ MPROG FOO.

Call ?RETURN 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 write uses all these calls. It uses ?GTMES to get a disk file name, then ?OPENs the disk file, creating it if necessary. Then it ?READs lines from the terminal and ?WRITEs the lines back to the terminal and to the disk file. It uses ?RETURN both to handle errors and to return normally to the CLI.

?OPEN

Open (optionally create and open) 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 error 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. For each file, you can use the same I/O packet for each call. Some parameters apply only to ?OPEN, others to ?READ/?WRITE, but it's easiest to set the whole thing up at the beginning and change only a few things -- like byte pointers -- at runtime.

A typical I/O packet (which we will use later in this chapter) specifies the following things:

- If the file exists, change its record format to that specified later in this packet (if needed);
- Open the file for appending, so that writing to it adds information to the end of the file, making it longer. (Or, you could delete and recreate the file, or overwrite the information in it.)
- Open the file for both input and output.
- Open the file with data-sensitive record format, which means that one or more special characters will be record delimiters.
- If the file does not exist, create it and open it; if it does exist, simply open it.
- No record can be longer than 120 characters. (This is ample in our case, since each record will be a line of text.)
- Reading or writing will start at the first record in the file.
- For the data-sensitive records (specified above), the delimiters will be the system defaults: NEW LINE character, form feed character, or null character.

All parameter packets should be built parametrically. This means that instead of numeric constants, 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 which, used with the .LOC pseudo-op, will automatically set up the correct locations.

The I/O packet we will use for ?OPEN, ?READ, and ?WRITE is ?IBLT words long. Figure 10-1 shows both the contents of each packet location *and* a sample packet for the terminal. In Figure 10-1, pkt-addr means packet-address.

One advantage of using symbolic offsets is that you can put them in any order. In Figure 10-1, the symbolic locations and the symbols that set bits within each location could be written in any order; e.g., pkt-addr+?ISTI could *follow* other offsets in the packet.

Name of Location	Contents
pkt-addr :	.BLK ?IBLT ; Reserve ?IBLT words for packet.
pkt-addr+?ISTI	?ICRF means adjust record format; +?IFCE means create file if needed; +?APND means open for appending; +?OFIO means open for input and output; +?RTDS means data-sensitive records.
pkt-addr+?IBAD	Byte pointer to I/O buffer; for example 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 a ?READ or ?WRITE.
pkt-addr+?IRCL	120. means maximum record length of 120 characters.
pkt-addr+?IFNP	Byte pointer to filename (pathname). For example, NAME*2 or CON*2.
pkt-addr+?IMRS	-1. Aside from setting this to -1, you can ignore it.
pkt-addr+?IDEL	-1. Aside from setting this to -1, you can ignore it.

```
; A real I/O packet, and filename and I/O buffer, looks like this:

FILE:   .BLK    ?IBLT                   ; Reserve ?IBLT words.
        .LOC    FILE+?ISTI              ; Location "I/o STatistics."
        ?ICRF+?APND?OFCE+?OFIO+?RTDS    ; Change record format+append+
                                        ;  +create+I&O+d-s recs.
        .LOC    FILE+?IBAD              ; Location "I/o Buffer ADdress".
        BUF*2                           ; Byte pointer to buffer.
        .LOC    FILE+?IRLR              ; Location "I/o Record Length Returned".
        0                               ; Sys returns no. of  bytes transferred.
        .LOC    FILE+?IFNP              ; Location "I/o FileName Pointer."
        .NAME*2                         ; Byte pointer to file (path) name.
        .LOC    FILE+?IMRS              ; Location "I/o MemoRy Size."
        -1                              ; Default with  -1.
        .LOC    FILE+?IDEL              ; Location "I/o" DELimiter".
        -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 10-1. Assembly Language I/O Packet

?READ and ?WRITE
Read and write 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 error return, AC0 contains an error code (described later in this chapter).

You issue ?READ to read a record, and ?WRITE to write to a record. When you first open a file, the system sets its file pointer to the beginning of the file. Each ?READ or ?WRITE operation transfers one record beginning at the current position of the file pointer; afterwards, the system positions the file pointer to the beginning of the next record.

For more on the packet, see the description given under the ?OPEN call.

?GTMES
Get 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 to start 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 separates command-line arguments. Thus, if the command you input to the CLI was

) X MYFILE/QQ=444 FOO ↓
the edited version of the message would be

MYFILE/QQ=444,FOO

MYFILE is argument number 0, /QQ=444 is argument 0's switch, and FOO is argument 1 (without switches).

Our example program later in this chapter uses GTMES to read the filename argument given in the XEQ command.

The ?GTMES parameter packet that we will use looks like:

```
pkt-addr: .BLK   ?GTLN    ; Reserve ?GTLN words.
          .LOC     pkt-addr+?GREQ    ; Location "Get Request".
          ?GARG          ; Contents: means "Get argument".
          .LOC     pkt-addr+?GNUM    ; Location "Get Number".
          1              ; Contents (means "Arg number 1").
          .LOC     pkt-addr+?GSW     ; Location "Get Switches".
          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
Terminate this process

Format:

?RETURN
exception (error) return

Input and Return:

Input: For a good (nonerror) return to the CLI, set AC2 to 0. For an error return to the CLI, load AC2 with ?RFEC+?RFCF+?RFER.

Return: Generally, none, because the program that issues ?RETURN is terminated.

This call terminates the calling process, and -- as described here -- returns control to the CLI. If you want the program to return normally, then simply set AC2 to 0 before issuing this call.

If the program is returning because of an error, load AC2 with ?RFEC+?RFCF+?RFER before issuing ?RETURN. The CLI will then interpret the error code passed in AC0 and print the appropiate error message on the screen.

This 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 shown. If one of them occurs, the system places the appropriate error number in AC0, then goes to the call's error 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 system call parameters. This usually means that you made a mistake with the packet and/or gave a bad byte pointer.

If you know that a call may encounter an error condition (as a ?READ may encounter 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, then could act on the result. For example:

```
        .
        ?READ   FILE  ; Read from file.
        JMP     EOF?  ; Error, check it.
        .             ; Normal return.
        .
        .
EOF?:   LDA     1, CODE  ; Get code for EOF.
                         ; (AC0 has current error).
        SUBC#   0,1,SZR  ; Skip if E.O.F.
        JMP     BYE      ; Other error, exit.
        JMP     CONT     ; Take other action if E.O.F.
CODE:   EREOF            ; MASM will insert correct
                         ; number.
```

We recommend that you use symbol mnemonics -- not constants -- wherever possible.

Example Program

The example program, entitled WRITE, is a short assembly language program that goes through the standard I/O cycle and 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.

Next, it types a prompt on the screen, reads a line from the screen, echoes the line on the screen, and writes the line to your specified file. Then it loops back to type the prompt again. When you type the characters NO↓ in response to 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 ?RETURN.

Figure 10-2 shows you a flowchart of this program's activity. Figure 10-3 shows 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 flowchart and program listing before proceeding.

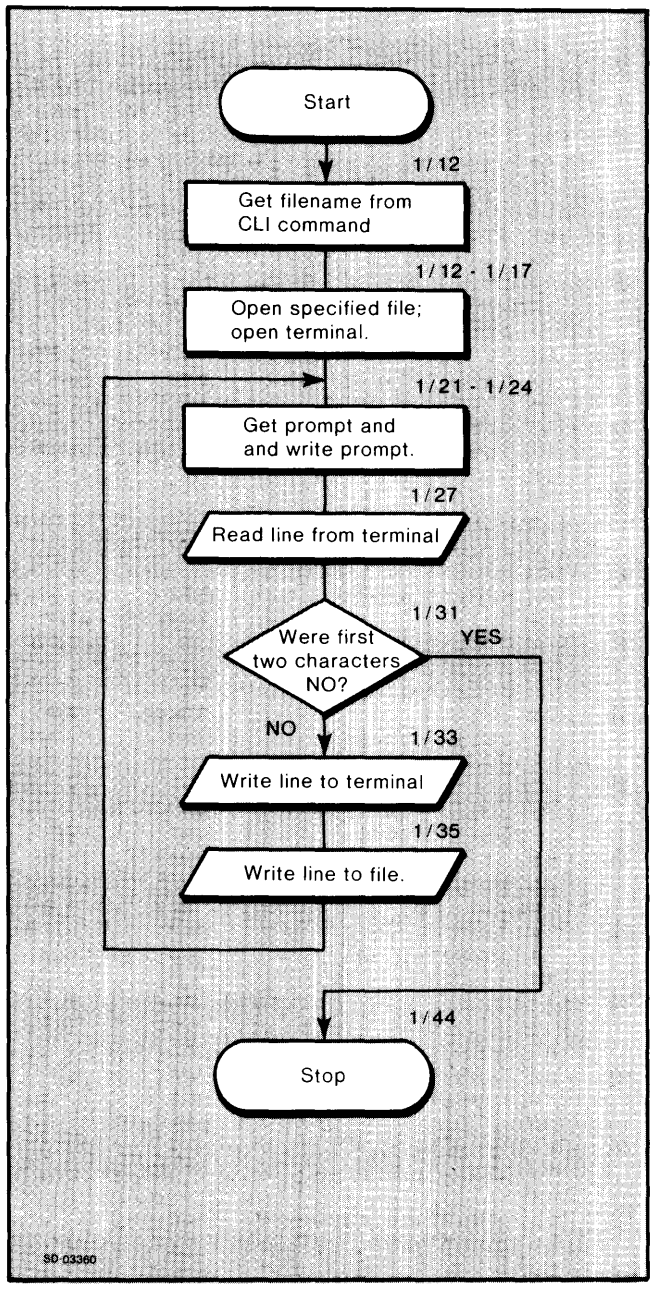


Figure 10-2. WRITE Flowchart

```
 0001  WRITE     AOS ASSEMBLER REV 03.20          13:22:10 01/09/81
 01                ;  THIS PROGRAM GETS A FILENAME FROM THE CLI XEQ COMMAND,
 02                ; OPENS THE TERMINAL AND THE SPECIFIED FILE, WRITES A PROMPT TO
 03                ; AND READS A LINE FROM THE TERMINAL, AND WRITES THE LINE TO
 04                ; TERMINAL AND FILE. TO STOP IT, TYPE NO<NEW LINE>.
 05
                           .TITLE  WRITE
 07                        .ENT    WRITE     ; FOR DEBUGGING
 08                        .NREL               ; UNSHARED.
 09
 10                ; GET FILENAME; OPEN TERMINAL AND SPECIFIED FILE.
 11
 12                WRITE:  ?GTMES  CLIMS     ; GET THE FILENAME.
 13 00004'000447           JMP     ERROR     ; ERROR, PROCESS IT
 14                        ?OPEN   FILE      ; OPEN FILE, CREATING IF NECESSARY.
 15 00011'000442           JMP     ERROR     ; ERROR, PROCESS IT.
 16                        ?OPEN   TERM      ; OPEN TERMINAL.
 17
 18
 19                ; WRITE PROMPT, READ LINE, CHECK FOR TERMINATOR, WRITE TO FILE.
 20
 21 00016'020444 LOOP:     LDA     0, PRMTP ; GET BYTE POINTER TO PROMPT.
 22 00017'040521           STA     0, TERM+?IBAD  ; PUT IN TERMINAL BUFFER ADDR.
 23                        ?WRITE  TERM      ; WRITE PROMPT TO TERMINAL.
 24 00024'000427           JMP     ERROR     ; ERROR, PROCESS IT.
 25 00025'020545           LDA     0, BUFFP ; GET BYTE POINTER TO I/O BUFFER.
 26 00026'040512           STA     0, TERM+?IBAD  ; PUT IN TERMINAL BUFFER ADDR.
 27                        ?READ   TERM      ; READ A LINE FROM TERMINAL.
 28 00033'000420           JMP     ERROR     ; ERROR, PROCESS IT.
 29 00034'020433           LDA     0, NO     ; PUT "NO" TERMINATOR IN AC0.
 30 00035'024536           LDA     1, BUFF   ; GET FIRST WORD (2 BYTES) OF BUFFER.
 31 00036'106475           SUBC #  0, 1, SNR ; SKIP IF NOT "NO".
   U00037'000000           JMP     BYE       ; USER TYPED "NO", EXIT.
 33                        ?WRITE  TERM      ; NOT "NO". ECHO LINE, CONTINUE.
 34 00044'000407           JMP     ERROR     ; ERROR, PROCESS.
 35                        ?WRITE  FILE      ; WRITE LINE TO FILE.
 36 00051'000402           JMP     ERROR     ; ERROR, PROCESS.
 37 00052'000744           JMP     LOOP      ; WRITE PROMPT AND DO IT ALL AGAIN.
 38
 39                ;  PROCESS ERROR AND/OR RETURN HERE.
 40
 41 00053'030406 ERROR:    LDA     2, FLAGS  ; GET ERROR FLAGS.
 42 00054'000402           JMP     .+2       ; SKIP SUBTRACTION FOR GOOD ?RETURN.
  FU00055'000000 BYE       SUB     2, 2      ; SET FOR GOOD RETURN.
 44                        ?RETURN           ; TO CLI.
 45 00060'000773           JMP     ERROR     ; ?RETURN ERROR.
 46 00061'150000 FLAGS:    ?RFEC+?RFCF+?RFER  ; ERROR FLAGS.
 47
 48 00062'000146"PRMTP:    PROMT*2           ; ADDRESS OF 1ST BYTE.
 49 00063'040547 PROMT:    .TXT  "Again?<12>"   ; PROMPT.
 50       060551
 51       067077
 52       005000
 53
 54 00067'047117 NO:       .TXT "NO"         ; TERMINATING CHARACTERS.
 55       000000
 56
 57                ;   ?GTMES PACKET TO GET THE MESSAGE.
 58
 59 00071'000004 CLIMS:    .BLK    ?GTLN   ; ?GTMES PACKET LENGTH.
 60       000071'          .LOC    CLIMS+?GREQ
```

Figure 10-3. Assembly Language WRITE Program with Errors (continues)

```
 0002  WRITE
 01 00071'000003          ?GARG    ; WE WANT TO GET AN ARGUMENT.
 02       000072'         .LOC     CLIMS+?GNUM
 03 00072'000001          1        ; ARGUMENT # 1 IS FILENAME (0 IS PROGNAME).
 04       000074'         .LOC     CLIMS+?GRES
 05 00074'000172"         NAME*2   ; NAME*2 IS BPTR TO NAME BUFFER.
 06       000075'         .LOC     CLIMS+?GTLN    ; END OF ?GTMES PACKET.
 07
 08 00075'000040 NAME:    .BLK     32.    ; RESERVE 64 CHARS FOR PATHNAME.
 09
 10              ; TERMINAL I/O PACKET.
 11
 12 00135'000014 TERM:    .BLK     ?IBLT    ; PACKET LENGTH.
 13       000136'         .LOC     TERM+?ISTI
 14 00136'040032          ?ICRF+?RTDS+?OFIO    ; CHANGE FMT+D-S RECORDS, I+O.
 15       000140'         .LOC     TERM+?IBAD
 16 00140'000366"         BUFF*2   ; BYTE POINTER TO I/O BUFFER ADDRESS.
 17       000142'         .LOC     TERM+?IRCL
 18 00142'000170          120.     ; MAX RECORD LENGTH OF 120 CHARS.
 19       000146'         .LOC     TERM+?IFNP
 20 00146'000322"         TNAME*2  ; BYTE POINTER TO TERMINAL FILENAME (@CONSOLE).
 21       000147'         .LOC     TERM+?IMRS
 22 00147'177777           -1      ; MEMORY BLOCK SIZE, DEFAULT.
 23       000150'         .LOC     TERM+?IDEL
 24 00150'177777          -1       ; D-S DELIMITER TABLE ADDR; DEFAULT.
 25       000151'         .LOC     TERM+?IBLT       ; END OF TERMINAL I/O PACKET.
 26
 27 00151'040103 TNAME:   .TXT "@CONSOLE"   ; GENERIC NAME FOR TERMINAL.
 28       047516
 29       051517
 30       046105
 31       000000
 32
 33              ; OPEN AND I/O PACKET FOR FILE.
 34
 35 00156'000014 FILE:    .BLK     ?IBLT    ; PACKET LENGTH.
 36       000157'         .LOC     FILE+?ISTI
 37 00157'040272          ?ICRF+?APND+?OFCE+?RTDS+?OFIO ; CHANGE FMT+APPEND+CREATE
 38                                ; +D-S RECS+I&O.
 39       000161'         .LOC     FILE+?IBAD
 40 00161'000366"         BUFF*2   ; BYTE POINTER TO I/O BUFFER (SAME AS TERM'S).
 41       000163'         .LOC     FILE+?IRCL
 42 00163'000170          120.     ; MAX RECORD LENGTH OF 120 CHARS.
 43       000164'         .LOC     FILE+?IRLR
 44 00164'000000          0        ; SYSTEM  RETURNS NO. OF CHARS TRANSFERRED.
 45       000167'         .LOC     FILE+?IFNP
 46 00167'000172"         NAME*2   ; BYTE POINTER TO NAME, OBTAINED BY ?GTMES.
 47       000170'         .LOC     FILE+?IMRS
 48 00170'177777           -1      ; MEMORY  BLOCK SIZE, DEFAULT.
 49       000171'         .LOC     FILE+?IDEL
 50 00171'177777          -1     ; D-S DELIMITER TABLE ADDR, DEFAULT.
 51       000172'          .LOC    FILE+?IBLT       ; END OF FILE PACKET.
 52
 53              ; I/O BUFFER.
 54
 55 00172'000366"BUFFP:   BUFF*2           ; ADDRESS OF 1ST BYTE.
 56 00173'000074 BUFF:    .BLK 60.         ; BUFFER OF 60. WORDS, 120. CHARS.
 57
  UU                      .END     .WRITE  ; START AT THE BEGINNING.

**00005 TOTAL ERRORS, 00000 FIRST PASS ERRORS
```

Figure 10-3. Assembly Language WRITE Program with Errors (concluded)

Writing and Assembling WRITE

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

```
) CREA/DIR  ASM ↵
) DIR  ASM ↵
)
```

Now, get into SED or SPEED:

```
XEQ  SED  WRITE.SR ↵
   or
XEQ  SPEED/D  WRITE.SR ↵
```

Type in the program according to Figure 10-3. This is an assembled listing, so *don't* type in any of the numbers or punctuation in columns 1-16. It might help you to pencil a straight line down the beginning of the instruction field (e.g., from the semicolons in the initial comments to the space before CLIMS: on the first page).

You can type in the program in either upper- or lowercase. One approach is to put comments and .TXT strings in lowercase, everything else in uppercase. The assembler outputs its listing file in all uppercase -- as you can see from Figure 10-3.

After typing in WRITE, leave the text editor and assemble it:

```
X  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 10-3. You will also get an assembler cross-reference, which we have omitted for brevity.

Analyzing Program WRITE

Let's analyze WRITE line by line. We will abbreviate listing references by page number/line number, e.g., 1/8 means "page 1, line 8". Notice that we did this convention in the flowchart to relate it with the program listing.

The program enters the starting address, WRITE (1/7), 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.

The .NREL pseudo-op, 1/8, specifies normal relocatable, unshared code.

The first group of code, found on 1/12-1/16, 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'll use to run the program. As for JMP ERROR, 1/13, we process all error conditions in this program by jumping to ERROR, described later.

The next section of code begins with the label LOOP: and extends from 1/21 through 1/37. This section gets the prompt, writes it to the screen, then reads from the screen. Then it checks the first two characters in the line to see if they are NO. WRITE does this by loading the first word (2 bytes, 2 characters) and comparing it to word NO that holds NO. If the words match, control goes to the next sequential instruction, which jumps to the BYE sequence and returns to the CLI. The JMP BYE on line 1/32 has a U (Undefined) error instead of a line number. If the first word in the buffer isn't the characters NO, the entire line is written to the screen and the file, and the sequence that begins with LOOP is repeated.

The error/return sequence extends from line 1/39 to 1/46. On an error, the program jumps to ERROR, which puts the appropriate flags in AC2 (1/41), then jumps over the SUB instruction to ?RETURN to the CLI. The flags tells 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 jumps straight to BYE (1/43, FU errors), which zeroes AC2 to indicate a normal return, then returns to the CLI.

Having reviewed the code, now let's examine the other locations and the parameter packets. Line 1/48 is a pointer to the first byte of the prompt, which follows. The prompt, 1/49-1/52, is the text string Again? followed by a NEW LINE character (<12>).

Line 1/54 is the terminating characters, NO, terminated by a null word of 000000 on 1/55.

The first parameter packet, CLIMS, 1/57-2/6, is needed by the ?GTMES call. We specified ?GARG in the first location because we wanted to get an argument (you can get many other things with ?GTMES). We specified the argument number, 1, in the next location, CLIMS+?GNUM. (Argument number 0 would be the program name, which we don't want here.) The next location, CLIMS+?GRES, has byte pointer NAME*2 to the space to receive the first argument.

NAME, on 2/8, will receive the file (path) name from the ?GTMES call. It has enough space for a pathname of 64 characters.

The next packet is the I/O packet for the terminal. It extends from 2/10 to 2/25. In location TERM+?ISTI, we say that we want to change record format if needed, use data-sensitive records, and that we want to open for Input and Output (?OFIO). In TERM+?IBAD, we have a byte pointer to the buffer BUFF; all the lines we type will be read into this buffer and written from it. Next, in TERM+?IRECL, we specify a maximum record length of 120 characters. This is more than enough, since each record will be a line of text typed on the screen. TERM+?IFNP has byte pointer TNAME*2 to the area that holds the filename. The last two locations 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 been wrong.

Following the TERM packet, we have the terminal's filename, @CONSOLE, lines 2/27 through 2/31. @CONSOLE is the generic name for the terminal input and output files and you can always use it in these situations.

Next, on lines 2/33 through 2/51, we have the I/O packet for the FILE. In location FILE+?ISTI, we say a lot: change format if needed, open for appending, create file the file if it doesn't exist, data-sensitive records, and open for input and output. In FILE+?IBAD, we have a byte pointer 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+?IRECL specifies the same record length as TERM+?IRECL: 120. In FILE+?IRLR, we place 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 TERM, the last two locations contain -1; otherwise, they don't concern us.

Line 2/55 has a pointer to the buffer BUFF. This is needed for the LDA back in line 1/25.

Line 2/56, reserves the buffer, BUFF, into which the program reads each line we type on the terminal. This buffer is 60 words or 120 bytes (characters) long.

Lastly, on line 2/58, we end the program with .END, directing control back to WRITE. The number is blown away by two U errors.

The assembler listing shows five errors, on lines 1/32, 1/43, and 2/58. As with error reports by other utilities, you cannot deduce the precise number of assembly errors from the number of error codes you receive. Take the error codes to be general indications of error conditions. Here, for example, all errors were caused by only two mistakes.

The first mistake is on line 1/43, where we forgot the colon after BYE. The second mistake involves our use of the symbol WRITE. We intended to use the symbol WRITE as a name for the start of the program. Deleting the period from the from the beginning of .WRITE should clear up this error.

Using SED or SPEED, fix the two erroneous lines. Change

```
BYE     SUB   2, 2      ; SET FOR...
   to
BYE:    SUB   2, 2      ; SET FOR...
```

and change

```
.END    .WRITE ; START AT....
   to
.END    WRITE  ; START AT ...
```

WRITE.SR with No Assembly Errors

Having corrected the assembly errors in WRITE.SR, we can reassemble it:

```
) X  MASM  WRITE ↵

 .TITLE     WRITE
)
```

MASM reports no errors, so we can proceed to Link WRITE:

```
) X LINK  WRITE ↵
LINK REVISION n  ON date  AT time
=WRITE.PR  CREATED
)
```

Like MASM, Link reports no errors. (Link errors are rare; if you ever receive one, and the text doesn't enable you to fix the error, consult the *Link User's Manual.)*

Will WRITE run right? Even though there are no assembly or Link errors, there may be some errors of logic or design in the program. We can now execute the program with the XEQ command. We need to include an output filename -- and, for the moment, choose FOO:

```
) X  WRITE  FOO ↵
```

Nothing. No prompt, no reaction. There appears to be a problem -- but let's type something anyway:

```
Something ↵
Something
Again?
```

Looks a little better. WRITE wrote back our line *and* displayed the correct prompt: *Again* . We don't know whether it wrote to disk file FOO yet. The prompt problem seems to have fixed itself. Try another line:

```
Something else ⤸
Something else
Again?
```

To check the disk file, FOO, we need to get back to the CLI. Type the terminating sequence, NO⤸:

```
NO ⤸
)
```

The terminator works! We're back in the CLI. Now, what about file FOO?

```
) TYPE  FOO⤸
Something
Something else

)
```

The disk file logic seems fine, too. Aside from the initial prompt problem, WRITE seems to be in pretty good shape. Check by running it again:

```
) X  WRITE ⤸
*ERROR*
NO SUCH ARGUMENT
ERROR FROM PROGRAM
X,WRITE
)
```

Hmmm -- we forgot the output filename, so the ?GTMES call bombed. Try it again with FOO:

```
) X  WRITE  FOO ⤸
```

Again, we receive no response on the first pass.

```
Sigh⤸
Sigh
Again?
```

Clearly, the initial prompt problem isn't going to go away. Stop WRITE and check FOO again from the CLI:

```
NO ⤸

) TY  FOO ⤸
Something
Something else
Sigh

)
```

Everything works except the initial prompt, which is nothing when it should be *Again?* . The next step is a very common one in assembly language programming: debugging.

Introduction to the Debugger

This section gives a Debugger overview. Read it before you proceed to operate the Debugger, as described in the next section, *Debugging WRITE.*

The Debugger is a system utility that lets you examine and make minor modifications to your program as you run it. We'll be using the following Debugger commands:

Command	What it Does
CR or ↑	Pressing the CR key tells the debugger to open and display the next location. An uparrow (usually SHIFT-6 keys) tells it to open and display the previous location.
addr:	Displays contents of addr. addr can be a symbol, a numeric address, or both; e.g., WRITE+2:
MODE x⤸	Changes the display mode. There are three modes that we'll be using: MODE N⤸ displays contents in instruction format; e.g., JMP ?USTA+3. MODE A⤸ displays contents in ASCII format;e.g., Ag. MODE F⤸ displays contents in octal; e.g., #000447. F is the default mode.
B addr⤸	Set a breakpoint at addr. Addr can be a period to specify the current location; e.g., B .⤸. You can use ⤸ or CR to set a breakpoint.
addr ; exp⤸	Changes to contents of location addr to exp.
SET #n;exp⤸	Set accumulator n to exp. n can be any of the accumulators; in our case, 0, 1, 2, or 3. Don't use spaces between arguments.
P	Proceed. P either starts the program or resumes execution from a breakpoint.
BYE	Leave the Debugger and return to the CLI.

When you display addresses, the Debugger treats the CR and NEW LINE characters differently. You must use CR to display the next location. Otherwise, you can use either CR or NEW LINE to terminate any Debugger command. (If your terminal lacks a CR key; use the RETURN key; if it lacks CR *and* RETURN, use the LINE FEED key.)

To debug programs, the debugger uses a program symbol table file, named **programname.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 plus sign (+); the Debugger prints + whenever it is ready to accept a command. Spaces and case are not important to the Debugger (e.g., it treats Write□+10: the same way as WRITE+10:).

The Debugger doesn't understand screen editing characters like CTRL-A. If you type something it doesn't recognize, it will display an error message. Generally, its error handling is sound, so you needn't worry about making mistakes.

For any location, only one Debugger mode gives a useful picture of the contents. If a location holds an instruction, like JMP or LDA, the appropriate mode is N, which displays in instruction format. The assembler listing can help in this process; for example, the listing in Figure 10-3 tells you that WRITE+17 contains an instruction, thus mode N is the appropriate mode for this location.

If a location holds a value, as many parameter packet locations do, the appropriate mode is F, the default mode. If a location holds ASCII characters, like **Again?**, the appropriate mode is A, which gives ASCII format. For a byte pointer, simply divide the location's value by 2 (e.g., 1264/2=) and check the location that the Debugger gives as a quotient.

Usually, you can tell which mode is correct because other modes will give surprising results.

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 in a program as a symbolic address; e.g., LOOP+3 or ERROR+1 if you used .ENT LOOP or ERROR. By using .ENT WRITE in WRITE, we named a symbolic location for use in debugger expressions.

Also, the start of each user program in unshared memory is known to the Debugger as ?USTA. Thus, even if we had not entered the symbol WRITE, we could have used ?USTA. The Debugger usually displays locations in relation to ?USTA. Still, it was a good idea to enter WRITE because it is familiar and it is easier to type than ?USTA.

In any case, to set a breakpoint at the beginning of WRITE, you'd issue the Debugger command
B WRITE⤸.

Do not set breakpoints within system calls. System calls, you remember, are macros that the assembler expands into several words. You may 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, do not set a breakpoint on the second word of a two-word instruction like ELDA. (We won't encounter this while debugging WRITE.)

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 colon. For example:

```
+  WRITE+16 :  020444
```

WRITE+16 is a Debug *address* or *addr*. (Remember that in examples we show user input and system prompts in **boldface** type and other system output in *italic* type.) To open and display successively higher locations, press the CR key (or, if your terminal lacks a CR key, the RETURN or LINE FEED key).

To display successively lower locations, use an uparrow (↑); usually SHIFT-6 on the keyboard. In debug dialogs, we show the CR key or equivalent as CR, the uparrow as ↑, and the NEW LINE character as ⤸, as usual. For example:

```
+  WRITE+16 :  020444  +  CR
?USTA+17: 040521  +  CR
?USTA+20: 172470  +  ↑
?USTA+17: 040521  +  ⤸
+
```

As you can see, the Debugger initially displays address as six-digit octal numbers, which don't often mean very much. This display mode is called mode F.

Mode N⤸ displays contents in instruction format and Mode A⤸ displays them in ASCII format. For example:

```
+  WRITE+17 :  040521  +  Mode N⤸
+  WRITE+17 :  STA  0  ?USTA+140  1⤸
+ mode A⤸
+  WRITE+17 :  AQ⤸
```

To change the contents of a memory location, first open and display its contents. Then type the new contents and NEW LINE ( ). The new contents can be an octal number, or it can be an instruction like MOV. Suppose you found that WRITE+200 contained zero, but you wanted it to contain MOVS 0,1 . The following sequence displays and changes this location:

```
+  WRITE+200:  000000 +  MOVS  0,1 ⤸
```

Changing the Contents of Accumulators or Carry

Before you start executing your program under the Debugger, or after you come to a breakpoint and are ready to execute a new segment of your program, you may want to change the contents of an accumulator or the state of the carry bit. The Debugger SET command lets you do this. The SET command has the format:

```
SET  x;expression
```

Codes you may use for x in the SET command are

#0	AC0 (Accumulator 0)
#1	AC1
#2	AC2
#3	AC3
#P	Program Counter
#C	Carry bit

Thus, for example, to place the value 177777 in AC3, you could issue the command

```
+  SET  #3;-1 ⤸
```

Starting or Continuing to Run Your Program

When you want to start or proceed with program execution, simply type P⤸. The Debugger will proceed at the address contained in #P, the program counter. Initially, #P contains the value specified by the .END pseudo-op in the assembly-language source program. If you are waiting at a breakpoint, #P contains the address at which execution stopped, so all you have to do is type P⤸ to continue where you left off. Sometimes, you may want to restart the program at an address other than the breakpoint. You can do this by updating #P with the SET command.

Ending a Debugging Session

When you are done debugging and want to return to the CLI, simply type

```
□BYE ⤸
```

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.

Debugging WRITE

When we left WRITE, we had a program with no assembly errors. It even ran properly, except for its first pass, when it displayed nothing instead of *Again?* for a prompt.

Now, having reviewed Debugger fundamentals, we can use the the Debugger to find out why WRITE doesn't give a prompt on its first pass through the code.

To start debugging the program, type

```
DEB  WRITE  FOO ⤸
```

The Debugger identifies itself and displays the initial contents of the accumulators:

```
AOS USER DEBUGGER, REV n
# 0= 000000 #1= 000000 #2= 000000 #3= 000000
+
```

You're ready to start debugging. The critical lines in the program assembler listing haven't changed, so you can use this as a guide. First, let's think about the problem. All system calls and logic in the program work, but the ?WRITE in line 1/23 (Figure 10-3) gives nothing at all, instead of the *Again?* prompt that line 1/49 specifies. After the program runs through the loop once, it gives the correct prompt. The problem must be somewhere around LOOP, because the prompt isn't written on the first pass.

For practice, we can look around location WRITE. WRITE specifies a system call, which the debugger will show as a sequence of 3 seemingly irrelevant instructions (beginning with ELEF,) followed by the JMP error return. For example, try the dialog in the left column.

```
+ write:  172470 +  ⤸                    In octal.
+ mode n ⤸                               Instruction
                                         mode.
+ write:  ELEF 2 ?USTA+7 1 + CR          These three are
?USTA+2: JSR @17 0  +  CR                all part of the
?USTA+3: COM# 0 0 SKP  +  CR             ⤸ ?GTMES call.
?USTA+4: JMP ?USTA+53  1  +  ⤸           Error return
                                         from
+                                        ?GTMES call.
```

So let's take a look around LOOP, which is where the problem is:

```
+ loop: EXPRESSION SYNTAX OR UNKNOWN SYMBOL ERROR
+
```

We didn't enter LOOP, so can't use it as a symbol name. Instead, let's use write+16:

```
+ write+16: LDA 0 ?USTA+62 1  + CR     Looks good.
?USTA+17: STA 0 ?USTA+140 1 + CR       So does this...
?USTA+20: ELEF 2 ?USTA+135 1 +↓        This is start
+                                      of another
                                       system call
                                       (1/23).
```

Let's take a look at the prompt, which begins at location write+63:

```
+ write+63:  STA 0 ?USTA+232 1 + ↓     Not useful...
+ mode a↓                              Use mode A.
+ write+63 :  Ag  +  CR                Here's Ag...
?USTA+64:  ai  +  CR                   and  ai...
?USTA+65:  n?  +  CR                   and  n?...
?USTA+66:  <012><000>  +  ↓            and  <null>.
+
```

Now, we're ready to set a few breakpoints. The problem seems to be somewhere in the LOOP, so we will set two breakpoints: one after LOOP at the STA at location WRITE+17 (line 1/22, another at the LDA at location WRITE+34 (line 1/29).

```
+ mode  n↓
+ write+17:  STA 0 ?USTA+140 1 +  B .  ↓
+ write+34:  LDA 0 ?USTA+67 1 +  B .  ↓
+ ?B↓
!0  ?USTA+17
!1  ?USTA+34
+
```

Here, we opened each location and checked its contents before setting the breakpoint with the B . command. Remember that . represents the current location, so it works as a debug address.

Having set two breakpoints, let's Proceed:

```
  P↓
!0  ?USTA+17
#0= 000003 #1= 177777 #2= 000604 #3= 000000
+
```

We've reached the first breakpoint, before the STA instruction that follows LOOP. Right away, something seems wrong: that *000003* in AC0 doesn't look much like a byte pointer. AC1 has some residual stuff from the ?GTMES call. AC2 has *604*, which must be the TERM packet address from the ?OPEN call. Let's verify that 604 is, in fact, the packet address:

```
+ 604: JMP ?ZMAX+20 0 + mode f↓   Back to mode
                                  F.
+ 604:  000071  +  CR             First loc. of
                                  packet.
?USTA+136 :040032  + ↓            This is the
+                                 ?ISTI word in
                                  the TERM
                                  packet.
```

Let's look at a packet location we specified (there are a number of locations we didn't specify in this TERM packet). From the listing, we know that .LOC TERM+?IFNP is a byte pointer to the filename. This location is at write+146.

```
□write+146 : 001440  +↓           Looks like a byte
                                  pointer.
+ 1440/2= 620                     Check by dividing by
                                  2.
                                  Symbol starts at 620.
+ 620:  040103 + mode a↓          Back to mode A.
+ 620:  @C + CR                   620 has @C...
?USTA+152: ON + CR                next location has ON...
?USTA+153: SO + CR                and next has SO...
?USTA+154: LE + CR                and next has LE.
?USTA+155:<0> <0> +↓              Name ends with a null.
+
```

So, in fact, ?TERM+?IFNP *does* contain a byte pointer to the the terminal's filename, @CONSOLE.

Let's proceed with the program, which should take us to the ?READ. We'll then need to type something for WRITE to read:

```
+ P↓                              Proceed.
Test1111↓                         Type Test1111↓

!1  ?USTA+34
#0= 001504 #1= 177777 #2= 000604 #3= 000000
+
```

We've reached the second breakpoint. AC0 looks as if it has a byte pointer now. This should be the byte pointer to the I/O buffer, and -- since the ?READ has executed -- we should be able to see the line we just typed *in* the buffer. You can check any byte pointer as we did above: by dividing the value by 2, then checking the address given as the quotient:

```
□1504/2= 642
+642:  Te  +  CR
?USTA+174:  st  +  CR
?USTA+175:  11  +  CR
?USTA+176:  11  +↓
+
```

Yes, the byte pointer in AC0 is correct now, and points to the I/O buffer. The buffer contains the string we typed: Test1111 . Let's proceed:

```
+ P↓
Test1111

!0  ?USTA+17
#0= 001264 #1= 052145 #2= 000625 #3= 000000
+
```

Because we didn't type NO↓, WRITE echoed the "Test" line and jumped back to LOOP. We're at the first breakpoint again. But, this time, AC0 has *1264* instead of the *3* that it had on the first pass. 1264 is probably the byte pointer to the prompt -- as it should be. Check it:

```
+ 1264/2= 532
+532:  Ag  +  CR
?USTA+64:  ai  +  CR
?USTA+65:  n?  +  CR
?USTA+66:  <012><000>  +
```

Okay, the correct byte pointer is in AC0. When we proceed, we should see it:

```
P↓

Again?
Test2222↓

!1  ?USTA+34
#0= 001504 #1= 052145 #2= 000604 #3= 000000
+
```

Just for practice, let's check the byte pointer in AC0 again; the buffer to which it points should have Test2222 in it.

```
□1504/2= 642↓
+642:  Te  +  CR
?USTA+174:  st  +  CR
?USTA+175:  22  +  CR
?USTA+176:  22 +
```

As expected, the buffer contains our last line: Test2222. Proceed again:

```
P↓
Test2222

!0  ?USTA+17
#0= 001264 #1= 052145 #2= 000625 #3= 000000
+
```

Again, because we didn't type NO↓ to WRITE, WRITE echoed the line and jumped back to LOOP; so we're at the first breakpoint.

The byte pointer in AC0 remains correct, which is consistent with our earlier experience: WRITE gave the correct prompt every time after the first pass. Clearly the loop is okay now.

The debug session has shown that WRITE skips the LDA in location LOOP for its first pass, but it executes this LDA correctly thereafter. Now, why does it do this?

Because we forgot the error return from the ?OPEN TERM call that occurs on line 1/16.

The error return is an essential part of each system call, but people can forget it. In this case, the system opens the terminal normally, but assumes that the LDA in line LOOP is the error return, thus the system skips it. When WRITE gets to the ?WRITE for the prompt, AC0 doesn't contain the prompt byte pointer, which is why we got nothing for the first prompt.

At the end of the loop, WRITE jumps back directly to LOOP, so the LDA executes properly after the first pass.

The solution is easy once you know the problem. For verification, use the debugger. First, get out of the debugger and restart it:

```
+ Bye↓

) DEB  WRITE  FOO↓

AOS USER DEBUGGER REV n
#0= 000000 #1= 000000 #2= 000000 #3= 000000

+
```

Set the critical breakpoint and run the program:

```
  B  Write+17↓
+ P↓

!0  ?USTA+17
#0= 000003 #1= 177777 #2= 000604 #3= 000000
+
```

Put the correct byte pointer value, 1264, in AC0, check the accumulators, and proceed; the prompt should appear as follows:

```
+ SET #0;1264 ↵
+ ?A ↵
#0= 001264 #1= 177777 #2= 000604 #3= 000000
+ P ↵
Again?
```

The right prompt appears on the first pass. The problem is definitely the missing error return, which you must insert with a text editor. Stop WRITE and get back to the CLI:

```
NO ↵
)
```

If you check file FOO (TYPE FOO), you'll find that the text lines typed during the debugging process were appended to it.

Final Version of WRITE

Using SED or SPEED, put instruction JMP ERROR after the ?OPEN TERM instruction. Don't insert an additional NEW LINE character if you want the listing line numbers to remain the same: simply put JMP ERROR on the proper line.

Then leave the editor, reassemble, and reLink WRITE:

```
) X  MASM/L=@LPT  WRITE ↵
.
) X  LINK  WRITE ↵
.
)
```

Figure 10-4 shows the final version of WRITE, again without the cross-reference. Boxed areas indicate changes from the initial version.

```
 0001  WRITE     AOS ASSEMBLER REV 03.20          14:21:33 01/09/81
01                ;  THIS PROGRAM GETS A FILENAME FROM THE CLI XEQ COMMAND,
02                ; OPENS THE TERMINAL AND THE SPECIFIED FILE, WRITES A PROMPT TO
03                ; AND READS A LINE FROM THE TERMINAL, AND WRITES THE LINE TO
04                ; TERMINAL AND FILE. TO STOP IT, TYPE NO<NEW LINE>.
05
                          .TITLE  WRITE
07                        .ENT    WRITE     ; FOR DEBUGGING
08                        .NREL             ; UNSHARED.
09
10                ; GET FILENAME; OPEN TERMINAL AND SPECIFIED FILE.
11
12                WRITE:  ?GTMES  CLIMS     ; GET THE FILENAME.
13 00004'000450           JMP     ERROR     ; ERROR, PROCESS IT
14                        ?OPEN   FILE      ; OPEN FILE, CREATING IF NECESSARY.
15 00011'000443           JMP     ERROR     ; ERROR, PROCESS IT.
16                        ?OPEN   TERM      ; OPEN TERMINAL.
17 00016'000436           JMP     ERROR
18
19                ; WRITE PROMPT, READ LINE, CHECK FOR TERMINATOR, WRITE TO FILE.
20
21 00017'020444 LOOP:     LDA     0, PRMTP ; GET BYTE POINTER TO PROMPT.
22 00020'040521           STA     0, TERM+?IBAD  ; PUT IN TERMINAL BUFFER ADDR.
23                        ?WRITE  TERM      ; WRITE PROMPT TO TERMINAL.
24 00025'000427           JMP     ERROR     ; ERROR, PROCESS IT.
25 00026'020545           LDA     0, BUFFP ; GET BYTE POINTER TO I/O BUFFER.
26 00027'040512           STA     0, TERM+?IBAD  ; PUT IN TERMINAL BUFFER ADDR.
27                        ?READ   TERM      ; READ A LINE FROM TERMINAL.
28 00034'000420           JMP     ERROR     ; ERROR, PROCESS IT.
29 00035'020433           LDA     0, NO     ; PUT "NO" TERMINATOR IN AC0.
30 00036'024536           LDA     1, BUFF   ; GET FIRST WORD (2 BYTES) OF BUFFER.
31 00037'106475           SUBC #  0, 1, SNR ; SKIP IF NOT "NO".
32 00040'000416           JMP     BYE       ; USER TYPED "NO", EXIT.
33                        ?WRITE  TERM      ; NOT "NO", ECHO LINE, CONTINUE.
34 00045'000407           JMP     ERROR     ; ERROR, PROCESS.
35                        ?WRITE  FILE      ; WRITE LINE TO FILE.
36 00052'000402           JMP     ERROR     ; ERROR, PROCESS.
37 00053'000744           JMP     LOOP      ; WRITE PROMPT AND DO IT ALL AGAIN.
38
39                ;  PROCESS ERROR AND/OR RETURN HERE.
40
41 00054'030406 ERROR:    LDA     2, FLAGS  ; GET ERROR FLAGS.
42 00055'000402           JMP     .+2       ; SKIP SUBTRACTION FOR GOOD ?RETURN.
43 00056'152400 BYE:      SUB     2, 2      ; SET FOR GOOD RETURN.
44                        ?RETURN           ; TO CLI.
45 00061'000773           JMP     ERROR     ; ?RETURN ERROR.
46 00062'150000 FLAGS:    ?RFEC+?RFCF+?RFER  ; ERROR FLAGS.
47
48 00063'000150"PRMTP:    PROMT*2           ; ADDRESS OF 1ST BYTE.
49 00064'040547 PROMT:    .TXT  "Again?<12>"   ; PROMPT.
50       060551
51       067077
52       005000
53
54 00070'047117 NO:       .TXT "NO"         ; TERMINATING CHARACTERS.
55       000000
56
57                ;   ?GTMES PACKET TO GET THE MESSAGE.
58
59 00072'000004 CLIMS:    .BLK    ?GTLN   ; ?GTMES PACKET LENGTH.
60       000072'          .LOC    CLIMS+?GREQ
```

Figure 10-4. assembly Language WRITE Program without Errors (continues)

```
 0002  WRITE
01 00072'000003          ?GARG   ; WE WANT TO GET AN ARGUMENT.
02       000073'         .LOC    CLIMS+?GNUM
03 00073'000001          1       ; ARGUMENT # 1 IS FILENAME (0 IS PROGNAME).
04       000075'         .LOC    CLIMS+?GRES
05 00075'000174"         NAME*2   ; NAME*2 IS BPTR TO NAME BUFFER.
06       000076'         .LOC    CLIMS+?GTLN    ; END OF ?GTMES PACKET.
07
08 00076'000040 NAME:    .BLK    32.   ; RESERVE 64 CHARS FOR PATHNAME.
09
10               ; TERMINAL I/O PACKET.
11
12 00136'000014 TERM:    .BLK    ?IBLT   ; PACKET LENGTH.
13       000137'         .LOC    TERM+?ISTI
14 00137'040032          ?ICRF+?RTDS+?OFIO   ; CHANGE FMT+D-S RECORDS, I+O.
15       000141'         .LOC    TERM+?IBAD
16 00141'000370"         BUFF*2  ; BYTE POINTER TO I/O BUFFER ADDRESS.
17       000143'         .LOC    TERM+?IRCL
18 00143'000170          120.    ; MAX RECORD LENGTH OF 120 CHARS.
19       000147'         .LOC    TERM+?IFNP
20 00147'000324"         TNAME*2 ; BYTE POINTER TO TERMINAL FILENAME (@CONSOLE).
21       000150'         .LOC    TERM+?IMRS
22 00150'177777           -1     ; MEMORY BLOCK SIZE, DEFAULT.
23       000151'         .LOC    TERM+?IDEL
24 00151'177777          -1      ; D-S DELIMITER TABLE ADDR; DEFAULT.
25       000152'         .LOC    TERM+?IBLT      ; END OF TERMINAL I/O PACKET.
26
27 00152'040103 TNAME:   .TXT "@CONSOLE"  ; GENERIC NAME FOR TERMINAL.
28       047516
29       051517
30       046105
31       000000
32
33               ; OPEN AND I/O PACKET FOR FILE.
34
35 00157'000014 FILE:    .BLK    ?IBLT   ; PACKET LENGTH.
36       000160'         .LOC    FILE+?ISTI
37 00160'040272          ?ICRF+?APND+?OFCE+?RTDS+?OFIO ; CHANGE FMT+APPEND+CREATE
38                               ; +D-S RECS+I&O.
39       000162'         .LOC    FILE+?IBAD
40 00162'000370"         BUFF*2  ; BYTE POINTER TO I/O BUFFER (SAME AS TERM'S).
41       000164'         .LOC    FILE+?IRCL
42 00164'000170          120.    ; MAX RECORD LENGTH OF 120 CHARS.
43       000165'         .LOC    FILE+?IRLR
44 00165'000000          0       ; SYSTEM  RETURNS NO. OF CHARS TRANSFERRED.
45       000170'         .LOC    FILE+?IFNP
46 00170'000174"         NAME*2  ; BYTE POINTER TO NAME, OBTAINED BY ?GTMES.
47       000171'         .LOC    FILE+?IMRS
48 00171'177777           -1     ; MEMORY  BLOCK SIZE, DEFAULT.
49       000172'         .LOC    FILE+?IDEL
50 00172'177777          -1     ; D-S DELIMITER TABLE ADDR, DEFAULT.
51       000173'          .LOC   FILE+?IBLT      ; END OF FILE PACKET.
52
53               ; I/O BUFFER.
54
55 00173'000370"BUFFP:   BUFF*2          ; ADDRESS OF 1ST BYTE.
56 00174'000074 BUFF:    .BLK 60.        ; BUFFER OF 60. WORDS, 120. CHARS.
57
58                       .END    WRITE   ; START AT THE BEGINNING.

**00000 TOTAL ERRORS, 00000 FIRST PASS ERRORS
```

Figure 10-4. assembly Language WRITE Program without Errors (concluded)

Running WRITE

WRITE should run properly now; we've spent enough time on it. Try it out:

```
) X  WRITE  FOO ↓
Again?
```

You've fixed it; the initial prompt is correct.

```
WRITE runs right, Q.E.D. ↓
WRITE runs right, Q.E.D.
Again?
NO ↓
)
```

Type FOO, and find that it contains all messages. WRITE is right, and all ventures into the text editor, MASM, Link, and the debugger have paid off. WRITE is a neat little program, and you can use it, or parts of it, to produce impromptu log files and to do other things.

Summary

Having debugged WRITE, you have completed the assembly language minicourse given in Chapters 9 and 10. You've read about the macroassembler, symbols, instructions, pseudo-ops, words, and bits. -- system calls, parameter packets, the program itself, and the debugger.

It's been a lot of work, but you've acquired a sound basic knowledge of DG assembly language programming. Congratulations.

What Next?

Chapters 1 through 5 gave you a sense of AOS and its text editors; you now have a working knowledge of these. From each language chapter you tried, you acquired a sense of how to use the language.

In sum, you now know how to use AOS. You can start writing your own programs, using other pertinent DG manuals.

End of Chapter

Glossary

This glossary describes computer-related terms that may be new to you -- either as words or in relation to DG's software products.

Advanced Operating System see AOS or AOS/VS

access control list a list of privileges, associated with every directory and file, that specifies the type of access allowed for any user.

accumulator (pertains to assembly language) a hardware register within the CPU of all Data General computers. Accumulators are used for arithmetic, value comparisons, and to load and store addresses. Each ECLIPSE MV-series machine has 4 fixed-point, 32-bit accumulators; each 16-bit ECLIPSE has 4 fixed-point, 16-bit accumulators. All MV-series, C-series, and some other ECLIPSE computers have four additional floating-point accumulators.

ALC instruction (pertains to assembly language) an Arithmetic-Logical-Carry instruction, like WMOV or MOV, used in arithmetic, bit manipulation, and value comparison.

ANSI American National Standards Institute, a committee that publishes standards for a large range of things, including computer languages and tapes, machine screws, and copiers.

AOS (Advanced Operating System) DG's Advanced Operating System for 16-bit ECLIPSE computers.

AOS/VS (Advanced Operating System, Virtual Storage) DG's Advanced Operating System for 32-bit ECLIPSE computers.

argument something that is acted upon by a command, statement, or instruction. For example, in QPRINT MYFILE, MYFILE is an argument to the QPRINT command. In PRINT *, "Hello" , both * and "Hello" are arguments to the PRINT statement.

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. All ASCII values fit in 7 bits.

backup files copied for safekeeping, usually onto magnetic tape, but sometimes onto other disks or diskettes.

batch the technique of processing in a continuous, noninteractive stream. Batch jobs are submitted to the system via the QBATCH command (described in the CLI manual) and processed by the system in one of four streams. Batch jobs do not require a terminal and can execute without user interaction (for example, overnight), and are suitable for big, well-organized things, like large sorts.

bit a Binary digIT. Can assume one of 2 values, 0 or 1. But 16 bits, as used in a DG computer word, can indicate 65,536 different numbers. 32 bits, in two computer words, can indicate over 4 billion numbers.

byte 8 bits, capable of storing one ASCII character (e.g., A) or 256 different numbers.

breakpoint a place where a debugger stops program execution during a debugging session. At a breakpoint, you can examine current values of variables (and, with assembly language, accumulators). DG debuggers allow you to set as many breakpoints as you need.

command in this book, a keyword that tells the CLI, text editor, or debugger what to do.

Command Line Interpreter (CLI) A system utility program whose commands allow interactive users to maintain files, to access all other utility programs, and to do many other things. The CLI replaces the JCL facility in batch-oriented systems.

compiler a utility program that translates statements in a high-level programming language (like FORTRAN or COBOL) into instructions the computer can understand.

control key see CTRL key

CRT (Cathode Ray Tube) a terminal with a keyboard and television-like video screen that displays characters.

CTRL key a key on the terminal keyboard that by itself does nothing but with other characters does a lot. CTRL sequences are used for screen editing/cursor control and to control system action.

console a device with a keyboard for input and a screen or printer for output. The filename is @CONSOLE. *In this manual we call it a terminal.*

cursor on a display screen, the cursor indicates the current position on a line. It is either a box superimposed on a character position, or a blinking underscore beneath a character position.

debugger a program 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. There are several debuggers, including SWAT for high-level languages and the assembly language debugger.

default, by default a value or parameter that a program uses if you do nothing about it. Two examples: the SED text editor displays line numbers by default; if you open a disk file, by default it is opened for both input and output.

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. (Also see root, user, UTIL.)

disk a fast mass storage device, consisting of one of more metal platters that rotate rapidly. The platter(s) have a magnetic coating that is written to and read from. One DG disk can hold up to 285 million bytes (characters). The operating system, all its directories and files, and all user directories and files are stored on disk.

file a collection of information stored as a unit, under a *filename*. Some device filenames are rigidly defined (e.g., @MTB0 for tape, @LPT for the line printer queue, @CONSOLE for the terminal); but disk filenames are flexible (also see names).

generic file a category of file, some of whose names you can set with CLI commands. For example, the generic list file can be set to @CONSOLE or a disk file.

I/O (Input/Output) The process of reading information into the computer's main memory (input) and/or writing information from memory (output). The input can come from, and the output go to: disk files, mag tape, card decks, the console, process I/O equipment, telephone lines, or microwave beams.

JCL Job Control Language, used in some systems to direct system operation for users.

line a sequence of ASCII characters that ends with either a NEW LINE, form feed, or null character.

line printer high speed printing device accessed by CLI QPRINT commands or by queue name @LPT.

macro a sequence of instructions or commands that can be called (accessed) by a single name; may or may not require arguments. Macros are primarily timesavers, allowing people to write a series of directives only once, then *call* it by one name. DG has a macroprocessor, MPL, for high-level languages as well as a macroassembler for assembly language; the CLI also provides for macros.

manager, system the person who plans and administers an operating system, deciding -- among other things -- who will be allowed to use the system and what privileges they will have.

MPL DG's macroprocessor for COBOL, FORTRAN 77, and PL/I. MPL allows users to insert a series of statements in their sources via a single reference.

name filenames can be from 1 to 31 characters, including letters, numbers, underscore (_), period (.), $, and ?. FORTRAN symbolic entity names can be from 1 to 32 characters including letters, numbers, and underscore, but must begin with a letter. COBOL names can be from 1 to 32 uppercase letters, numbers, and dash (-). Extended BASIC symbol names can be a letter or letter and number, optionally followed by $ to indicate a string variable. Assembly language symbol names, by default, are unique to only five characters and can include letters, numbers, period (.), $, and ?; they must begin with a letter.

offset in assembly language, a location relative to another symbol; e.g., TABLE+1 or CON+?IBAD.

operator, system the person who physically operates a system for users.

page in text editing, the number of lines between form feed (CTRL-L) characters; the text editors have paginating commands not described in this book. In terms of computer memory/disk I/O, a page is 2,048 bytes (characters).

password a combination of letters that, used in conjunction with your username, allows you to log on and use the system.

pathname a path, usually including directory names, to a directory or individual file. For example, :UDD:JACK:LEARNING:MYFILE.

peripherals directory \ (PER) \\ the system directory that holds all device entries; full pathname :PER or the prefix @. The prefix @ that you use with devicenames and queuenames specifies the periperals directory.

process an executing program.

program a series of instructions, translated into binary codes, that the computer can execute. The text editor that allows you to write the instructions, the compiler or assembler that translates them, and the Linker that positions them correctly are themselves programs. So are the CLI and operating system. Each program file that you can execute under the operating system ends in the characters .PR .

record a series of 1 or more characters written to or read from a file. There are four record formats: data sensitive (delimited by NEW LINE, form feed, or null); fixed (established to be a constant number of characters); variable (established by the number of characters written); and dynamic (established by the variable involved in the write or read).

root directory (:) the system master directory that both contains and gives access to all other directories.

search list a list of directories that the CLI will scan whenever it can't find the specified program or macro in the working directory; established with SEARCHLIST command.

source file the source code written by a programmer. It's compiled into an object file which in turn is linked to form an executable program file.

SWAT DG's high-level language debugger, which works with FORTRAN 77 and PL/I.

symbol the name that identifies some procedure, variable, array name, or location. Often created by users, but sometimes defined by the language or system. For legal names, see names.

tape a magnetic medium suitable for file backup and mass storage. Tape drive device names are @MTBn, @MTAn, or @MTCn, where n is the number dialed on the drive thumbwheel (for MTB and MTA drives) or selected during tape installation (for MTC drives). Tape *files* are numbered sequentially from 0. With a tape on drive MTB0, you could access the second file as @MTB0:1 .

terminal a device with a keyboard for input and a screen or printer for output. The filename is @CONSOLE.

user

directory the directory created and maintained for each interactive user. Usually becomes the working directory when you log on. Allows inferior directories.

directory directory (:UDD) the system directory that contains each interactive user directory.

profile a disk file, created by the system manager or someone else in authority, that contains each user's username, password, disk space allowance, and other privilege specifications.

user, system anyone who -- in any capacity -- uses the system. Can be manager or operator seeking information, programmer, or nontechnical person.

username the name under which a system user logs on; usually assigned by the system manager. The username is also the name of the user directory.

UTIL the utilities directory. It contains most (if not all) utility programs on the system; it is also often found in search lists. Its full pathname is :UTIL .

utility, utility program a program supplied by DG to help you develop your own programs; e.g., compilers, Link, debugger. Some utilities are shipped as part of the operating system; others are optional.

word in memory, a 16-bit entity. In text, a sequence of one or more ASCII characters that begins and ends with either a blank, NEW LINE, form feed, or null.

working directory the directory where you currently are: the current directory.

Index

Within this index, the letter "f" means "and the following page"; "ff" means "and the following pages". For each topic, primary page references are listed first. All letters are lowercase, except for program names (e.g., CLI, FORTRAN), commands (e.g., ACL, APPEND), and programming language statements, instructions, and psuedo-ops.

For *definitions* of keywords, see the preceding glossary. Glossary entries generally aren't included here.

A

B

C

D

E

F

G

H

Q

R

S

T

U

V

W

X

Z

Data General

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government
- ☐ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify) _____		
_____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ MP/OS
- ☐ RDOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ HASP
- ☐ RJE80
- ☐ RCX 70
- ☐ CAM
- ☐ XODIAC
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

CUT ALONG DOTTED LINE

Data General

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

DataGeneral

TP_____

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

SHIP TO: (if different)
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #_____
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No._____ Expiration Date_____

☐ Purchase Order Number:_____

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other _____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order _____ Phone _____ Extension _____

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

Title

DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

User Documentation Remarks Form

Your Name _____ Your Title _____

Company _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title _____ Manual No. _____

Who are you? ☐EDP Manager ☐Analyst/Programmer ☐Other _____
☐Senior Systems Analyst ☐Operator _____

What programming language(s) do you use? _____

How do you use this manual? *(List in order: 1 = Primary Use)* _____

___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide _____

About the manual:	Yes	Somewhat	No
Is it easy to read?	☐	☐	☐
Is it easy to understand?	☐	☐	☐
Are the topics logically organized?	☐	☐	☐
Is the technical information accurate?	☐	☐	☐
Can you easily find what you want?	☐	☐	☐
Does it tell you everything you need to know	☐	☐	☐
Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580

069-000018-02