Learning to Use Your RDOS/DOS System

069-000022-01

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. 069-000022

Printed in the United States of America
Revision 01, August 1979

NOTICE

Data General Corporation (DGC) currently does not supply double-density diskette units, but may possibly supply them in the future. DGC currently does supply single-density diskette units. This manual contains references to both double- and single-density diskette units, but you should ignore references to double-density units unless and until DGC supplies such units. In no event should you assume that DGC will eventually supply double-density diskette units.

Learning to Use Your RDOS/DOS System
069-000022

Revision History:

Original Release - September 1978
First Revision - August 1979

A vertical bar or an asterisk in the margin of a page indicates substantive technical change or deletion, respectively, from revision 00.

Preface

RDOS is an acronym for Data General's Real-time Disk Operating System; DOS stands for Disk Operating System. In an hour or two, you can develop a working sense of either system by using it. This book leads you through the steps required to:

- Talk to the system through the Command Line Interpreter (CLI);
- Write programs with a text editor program;
- Produce and run a FORTRAN IV program;
- Program in Extended BASIC; and
- Write, produce debug, and execute an assembly language program.

We don't attempt to describe all features of your operating system, the CLI, other utility programs, or the compilers you'll be using. These are all described completely in the manuals listed below.

If your operating system is new to you, this book will give you a practical basis for using it. If you want to *generate* a system (which someone must do before you can really use this manual), see *How to Load and Generate Your RDOS System* for RDOS, or the appropriate chapter of *The DOS Reference Manual* for DOS. After you have generated a system, return to this manual.

If you plan to program in an advanced language other than FORTRAN IV or BASIC, you'll be using a different compiler, but the material on the CLI and assembly language will be useful nonetheless.

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: $\{\text{required}_1 \mid \text{required}_2\}$ 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 RETURN 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.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

R is the RDOS/DOS CLI prompt.

Related Manuals

We list these manuals by name and number here; you can find them described in context as you read this book.

Extended Assembler User's Manual, 093-000040 or 093-000139
basic BASIC, 093-000088
Extended BASIC User's Manual, 093-000065
Extended BASIC System Manager's Guide, 093-000119
Command Line Interpreter User's Manual, 093-000109
Symbolic Debugger, 093-000044
DOS Reference Manual, 093-000201
FORTRAN IV User's Manual, 093-000053
FORTRAN IV Runtime Library, 093-000068
FORTRAN Commercial Subroutine, 093-000107
How to Load and Generate Your RDOS System, 093-000188
Extended Relocatable Loaders User's Manual, 093-000080
Macroassembler User's Manual, 093-000081
Introduction to Programming the NOVA™ Computers, 093-000067
Introduction to RDOS, 069-000002 (A technical primer)
RDOS Reference Manual, 093-000075 (the primary technical book on RDOS)
RDOS/DOS User's Handbook, 093-000105 (a handy summary of RDOS and DOS utility commands and errors)
Superedit Text Editor User's Manual (RDOS/DOS), 093-000111
How to Generate Your DOS System, 093-000222

End of Preface

Contents

Chapter 1 - Terms and Concepts

Chapter 2 - At the Console - RDOS

Chapter 3 - At the Concole - DOS

Chapter 4 - Common CLI Commands

Chapter 5 - Creating and Editing Text

Chapter 6 - Instant FORTRAN IV Programming

Chapter 7 - Extended BASIC Programming

Chapter 8 - Assembly - Language Programming: The Assemblers

Chapter 9 - Programming RDOS or DOS Assembly Language System Calls

Tables

Illustrations

Chapter 1
Terms and Concepts

What are RDOS and DOS?

The Real-time Disk Operating System (RDOS) is a general-purpose software package that can support real-time control, batch, or program development. RDOS runs on NOVA® and ECLIPSE® computers; it can use many different kinds of disks, and supports up to 512K bytes of memory.

The Disk Operating System (DOS) is a compatible subset of RDOS, and supports up to 64K bytes of memory. DOS runs on diskettes or hard disks, on NOVA and microNOVA™ computers.

Both RDOS and DOS permit multitasking. This means that different program tasks can run 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 waiting for a slow peripheral device to complete an operation. However, multitasking is outside the scope of this book. You can find more information on multitasking in Chapter 5 of your RDOS or DOS Reference Manual.

How Do I Develop Programs?

Depending on your programming language interests, you'll use one or more system programs or utilities (two terms we use interchangeably). You'll always need the CLI, whose operating procedures we describe in Chapters 2 and 3, and whose common commands we survey in Chapter 4.

Unless you're a BASIC programmer, you'll be using a text editor utility to write your programs. Chapter 5 explains Superedit - one of our editors.

If you are a FORTRAN IV programmer, this is the cycle you will follow from creating through executing a program:

1) Create a FORTRAN source file using Superedit.

2) Compile the source file; compilation produces a binary file.

3) Load the binary file with the required FORTRAN libraries; the result is an executable program.

4) Execute the program.

Chapter 6 leads you through the steps to produce and run a sample FORTRAN program.

If you are a BASIC programmer, you need no system utilities except the CLI. Chapter 7 gives a capsule sketch of BASIC programming.

If you are an assembly language programmer, your procedure is much the same as the FORTRAN programmer's:

1) Create an assembly language source file using Superedit.

2) Assemble the source file; the assembly produces a binary file. Chapter 8 describes the assemblers.

3) Load the binary file. This produces an executable program.

4) Execute the program.

Chapter 9 leads you through the procedures to create and execute an assembly language program.

The remainder of this chapter describes two fundamental system concepts: memory management and files.

Managing Main Memory

RDOS memory can be divided into two program *grounds*, called foreground and background; each ground can run a program. Memory can be mapped by a hardware map, which allows each program to use up to 32K of memory, or it can be unmapped and allow each to use somewhat less memory. DOS memory can run only one program at a time.

Both systems provide several methods to increase the effective size of a program's address space: program swapping and overlays. Swapping and overlays allow several program segments to occupy the same area of main memory at different times. Since each area receives different code at different times, you can use a given memory area more efficiently.

The details on mapping, swapping and overlays, however, are outside the scope of this manual. Chapters 3 through 6 of the *RDOS Reference Manual*, and Chapters 3 through 5 of the *DOS Reference Manual*, explore them further.

What is a File?

A *file* is a collection of information treated as a unit. This data can be discrete scraps of data, like April's sales figures or December's mortgage balances; or it can be unified program code which does useful work, such as sorting other data or computing interest. The first type of file is a *data* file and the second a program or *save* file. Certain save files can read data files, act on what they read, and produce new, updated data files. Each utility program and compiler you received from Data General - including the Command Line Interpreter Program (CLI) and the operating system itself - is a save file.

Files are stored on devices such as magnetic tapes and disks. Disk files are the most versatile, and your system has a way of structuring and managing them. A disk file can be very large or very small; the largest file can store millions of bytes; the smallest, 0 bytes.

You give a *filename* to each disk file to identify it. Filenames have from 1 to 10 of the following ASCII characters: upper- and lowercase letters, numbers, or dollar sign ($). In filenames, upper- and lowercase letters are considered identical; for example, the system sees no difference between filenames TEST and Test. A disk filename can also have an "extension", which is a period followed by one or two legal filename characters; for example, TEST.FR.

You can use any legal filenames and extensions for your files. But be aware that the system and its utility programs follow these conventions for filename extensions:

- It requires CLI macro files to have the extension *.MC*.
- It assumes (but does not require) that assembly language source files have the extension *.SR*.
- It assumes (but does not require) that FORTRAN source files have the extension *.FR*.
- It assigns the extension *.RB* to the binary version of each source filename after compiling or assembling it.
- It assigns executable program files the extension *.SV* (save) and overlay files the extension *.OL* after processing them.
- It assigns the extension *.DR* to user directory names when you create the directories.
- It assumes (but does not require) that library files have the extension *.LB*.
- It uses the extension *.CM* for some temporary CLI files.

We touch on all of these file types in this book.

After you've created a few files, you'll need some way to keep track of their names. You may even want to group your files into logical sets. RDOS and DOS provide a mechanism called a *directory* to help you keep track of your files and to categorize them. A directory file contains and catalogs information used to access other files and devices. It's similar to the table of contents in a book.

Each system allows you to create your own directories. You'll learn about creating and managing directories in the next chapters.

Any file can be *permanent* or *nonpermanent*. Permanent files cannot be accidentally deleted but require special treatment in some CLI commands. All system files and files you create in this book will be nonpermanent. Later on, for more on this, consult the CHATR command in the CLI manual.

Let's Get Started

If yours in an RDOS system, proceed to the next chapter; if you're running DOS, skip to Chapter 3.

End of Chapter

069-000022-01

Chapter 2
At the Console - RDOS

This chapter describes a sample session with RDOS and the CLI, and leads you through many of the things you do on the keyboard to create and organize files. When you've finished the session, you'll have a practical working knowledge of RDOS basics. (For a session with DOS, see Chapter 3.)

If you inadvertently depart from a step we describe, your later experiences may differ from the text description. This is ok; mistakes are a primary vehicle for learning, and the CLI will usually prevent disasters. If you get really lost, go back to the beginning of the section you are in and give the files and directories different names; for example, FILEA1.MC instead of FILEA.MC.

Later on, you can find more detail on the features and ramifications of your commands in the *RDOS/DOS Command Line Interpreter User's Manual* or the *RDOS Reference Manual*. If you generated your own RDOS system, you already have some experience with RDOS.

Starting Up (Bootstrapping)

Turn your system console ON, and make sure it is ON LINE. On some DASHER™ displays, the LINE switch is in back of the console. If this console has both upper- and lowercase letters, set it in uppercase mode with the ALPHA LOCK key because the program that brings up the system doesn't accept lowercase letters. Later you can change back to upper- and lowercase mode.

Press or turn the computer's POWER switch to ON and flip the disk power switch (not the LOAD/READY or START switch) to ON. If your system is on a removable disk and the diskette is not in its drive, insert it in drive 0 and flip the LOAD/READY switch to READY (or press START). On any disk, wait for the READY light.

Now, turn to the system console. If it shows an exclamation point (!) prompt, then you have a virtual or *programmed* console. If it shows nothing, then your computer has hardware data switches.

For a machine with a programmed console, find nn in Table 2-1. Type 1000nnL (for example, 100033L) on the *system* console next to the ! prompt. Skip the next paragraph.

Table 2-1. Disk Device Codes

Disk Type Model No.	Device Code (octal) nn =	Set These Switches Up (Others Down):*
Fixed-head		
Model 6001-08	20	0, 11
Model 6063/64	26	0, 11, 13, 14
Moving-head		
Model 6060/61	27	0, 11, 13, 14, 15
All others	33	0, 11, 12, 14, 15

* For a disk on the second controller, also set switch 10 up

For a machine with hardware data switches, turn to the computer front panel and make sure the data switches are set properly for your type of disk, as shown in Table 2-1. The data switches are toggle-type switches, numbered 0 through 15. If the switches aren't set properly, fix them. (If your computer lacks automatic program load, see Chapter 2 of *How to Load and Generate Your RDOS System* for the manual load procedure.) Now, lift the RESET switch, then the PROGRAM LOAD switch.

For either type of computer, the system console will ask:

FILENAME?

It's asking for the name of your RDOS system. If the person who generated this system gave it any name other than SYS, you must type that name; but let's assume it was named SYS. To indicate SYS, type SYS and press the RETURN key (), or simply press).

FILENAME?)
RDOS REV x.xx (x.xx is the revision number)
DATE (M/D/Y)?

(If you receive the message *FILE NOT FOUND: SYS.SV*, try to discover (or remember) the system's name; then type this name and).)

Answer the date/time questions:

FILENAME?)
RDOS REV x.xx
DATE (M/D/Y)? 4 2 79)
TIME (H:M:S)? 13 10) (Hours in 24)
R

R is the CLI prompt; it means that the CLI is ready to accept a command. You'll see it hundreds of times in the future. You can now switch to upper- and lowercase, if this applies. RDOS will translate lowercase letters to uppercase.

Typing Mistakes, Control Characters and BREAK Key

If you type a faulty command and press) without realizing that you've typed in a faulty command, the CLI will respond with an error message - usually *FILE DOES NOT EXIST: faulty-entry* - and will return the R prompt. If you want to correct a line before pressing return, simply press the DEL or RUBOUT key sequentially to erase characters from right to left. (On printer consoles DEL or RUBOUT echoes a backarrow (←) or underscore (_) for each character erased.) You can erase an entire line by typing backslash (\) or SHIFT-L.

To stop execution of any CLI command or utility program (except an editor), press the CTRL key, hold it down, then press A. CTRL-A halts the command or program and returns the R prompt.

You can interrupt output to your console at any time by typing CTRL-S. The output won't be lost; you can continue it from the point of CTRL-S by typing CTRL-Q. You can use CTRL-S and CTRL-Q intermittently to read a long file on a display console.

If your computer has a programmed console (! prompt), the BREAK key on the system console keyboard stops execution of all software. To continue with the program that was running, type P) (uppercase P, then)). We mention this here because, in the course of bringing up and running RDOS, you may inadvertently hit the BREAK key. If everything stops and the system console shows !, try typing P).

The Session

If your system console has a display screen instead of a printer, you should record console dialog in the disk log file so that you can print and review it later. If the system console has a printer, it provides reviewable copy as you type; you don't need a disk log file and can proceed to the section called *Master Directory Name.*

Log File

With a display terminal, type

```
LIST LOG.CM)
```

If there is an old LOG.CM log file on the disk, the screen will display

LOG.CM

and a number and letter, then the R prompt. Preserve the old log file by typing

```
RENAME LOG.CM LOGOLD.CM)
R
```

Next, whether or not there was an old log file, type

```
LOG/H)
```

to create a new log file, open it, and start recording console dialog in it. Later, to close the log file, you'll type ENDLOG).

Master Directory Name

Next you should discover your master directory name, so type:

```
MDIR)
```

The CLI returns the master directory name, which varies with the type of disk you have. Generally, it is either DP0, DZ0, or DS0. We use *Dxx* to indicate the master directory name, so you should mentally substitute the name returned from MDIR) for *Dxx* in this session.

Creating Some Files

The next step, naturally, is to create a file. The CLI offers several file-creating commands, and we choose CRAND:

```
CRAND FILEA)
R
```

Voila. You now have an empty disk file named FILEA. FILEA was an *argument* to your command; it told the CLI what to name the new file. Some CLI commands require arguments, others don't.

You can verify FILEA's existence with the LIST command:

```
LIST FILEA)
FILEA.   0 D
R
```

and remove FILEA with the DELETE command:

```
DELETE/V FILEA)
DELETED FILEA
R
```

The /V is a *switch*, which modifies the basic meaning of a command; here, it told the CLI to verify the deletion. Now, create a different version of FILEA:

```
CRAND FILEA.MC)
R
```

Now there's an empty FILEA.MC on the disk. Normally, you'd use a text editor to put something in it, but you haven't reached Chapter 5 -- *Creating and Editing Text* -- yet. You can, however, insert text by transferring it directly from the console into FILEA.MC:

```
XFER/A/B $TTI FILEA.MC
MESSAGE HELLO)
CTRL-Z        (Hold down CTRL key and press Z)
R
```

The CTRL and Z keys won't echo as characters on the console. On a display terminal, the R prompt may leap to the top of the screen when you type CTRL-Z. If so, type CTRL-L to clear the screen and proceed with a clear screen.

You've just transferred the CLI command MESSAGE and the text string HELLO into FILEA.MC. The XFER command transfers the contents of one file (the console input file, $TTI) to another file (disk file FILEA.MC). XFER requires two arguments; in this case, they were $TTI and FILEA.MC. The /A switch specifies ASCII transfer; /B tells XFER to append to the existing file. If you omitted /B, XFER would try to create FILEA.MC, which would produce an error message because FILEA.MC already exists.

Try to execute FILEA.MC:

```
FILEA)
HELLO
R
```

Your FILEA.MC is a CLI macro. The .MC extension tells the CLI to execute all commands within it. You can omit the .MC extension to *execute* a file, but you must include it for all other commands that involve the file. Check FILEA.MC's statistics with LIST:

```
LIST/E FILEA.MC)
FILEA.MC 14 D 04/02/79 13:20 04/02/79 [000444]0
R
```

the /E switch tells the CLI to list every statistic about the file. These include byte-length (14), organization type (D means random), date and time created or last modified, date last opened, starting disk block address in octal, and use count. Obviously, the central four categories will differ for your own FILEA.MC.

To check the file's contents, type:

```
TYPE FILEA.MC)
MESSAGE HELLO
R
```

Now, you can add a little sophistication to FILEA.MC:

```
XFER/A/B $TTI FILEA.MC)
MESSAGE HOW MANY BLOCKS ARE LEFT)
MESSAGE ON THIS DISK? ;DISK)
MESSAGE WHAT'S THE CURRENT DIRECTORY)
GDIR)
CTRL-Z
R
```

One problem with inserting text this way is that you can't edit. If you make a mistake and don't correct it before you terminate the line with a),you must delete the file and rebuild it from the beginning. Having made no mistakes, check the improved macro:

```
FILEA)
HELLO
HOW MANY BLOCKS ARE LEFT
ON THIS DISK?
LEFT:9008 USED:768
WHAT'S THE CURRENT DIRECTORY?
Dxx
R
```

FILEA.MC is much larger now, as you can see:

```
LIST FILEA.MC)
FILEA.MC 118 D
R
```

FILEA.MC now contains four MESSAGE commands, a DISK command, and a GDIR command. Each would work separately, but you've combined them. (Your own DISK figures may differ from those here).

Proceed to create another macro:

```
XFER/A $TTI FILEB)
TYPE FILEA.MC)
CTRL-Z
R
```

Here, FILEB doesn't exist, so XFER will create it, and you can omit the /B switch. By doing so, you saved two steps, the CRAND FILEB and the /B switch, but you've been a little careless:

```
FILEB)
FILE DOES NOT EXIST: FILEB.SV
R
```

The CLI looked for FILEB.MC, then for the save file, FILEB.SV; finding neither, it returned the error message. To fix it, give FILEB the .MC extension:

```
RENAME FILEB FILEB.MC)
R
FILEB)
MESSAGE HELLO
MESSAGE HOW MANY BLOCKS ARE LEFT
MESSAGE ON THIS DISK: DISK
    .
    .
    .
R
```

You planned to have your second file type your first file, and you succeeded. Now you'd like to compare the files, so you type LIST/E without an argument. See Figure 2-1.

Your command told the CLI to list all nonpermanent files in the master directory; who knows how long the listing would have continued if you hadn't hit the CTRL-A keys to interrupt the command and return the CLI prompt. (Your own listing will differ from that above.) The letters following the byte count indicate file type and organization.

There are some old friends in Figure 2-1: SYS.SV, CLI.SV (save files always end in .SV), and FILEA.MC. If you generated your own RDOS system, you'll recognize BOOT.SV. Unfortunately, FILEB.MC didn't show up. You could list the entire contents of directory Dxx to find FILEB.MC, but there's a better way: create a directory for the two macro files.

Creating Some Directories

There are two kinds of directories you can create: *secondary partitions* (whose size cannot change) and *subdirectories* (which are flexible). You decide on a subdirectory:

```
CDIR MACRODIR)
R
```

Now, to move the macros into the new directory:

```
MOVE/V MACRODIR FILEA.MC FILEB.MC)
FILEA.MC
FILEB.MC
R
```

```
LIST/E)
SYS.SV      36864  SD    03/26/79  13:36  03/26/79  [002707]  0
DSKED.SV    18432  SD    03/29/78  16:18  03/29/78  [001670]  0
RLDR.SV      4608  SD    13/23/78  13:16  03/23/78  [015741]  0
CLI.OL      43008  C     10/05/78  10:14  03/30/79  [000366]  1
BOOT.SV      6656  SD    10/05/78  23:03  10/05/78  [000522]  0
FILEA.MC      118  D     04/02/79  13:24  04/02/79  [000444]  0
CLI.SV      10752  SD    10/05/78  10:14  10/05/78  [003467]  0
CTRL-A
INT
R
```

Figure 2-1. Master Directory Listing

The MOVE command copies the files into the new directory; the /V switch instructs the CLI to verify their names as they arrive. Now, the files are safely in MACRODIR; you can make MACRODIR the current directory and then list the files in it:

```
DIR MACRODIR)
R
LIST)
FILEA.MC  118 D
FILEB.MC  14
R
```

The DIR command makes MACRODIR the current directory. DIR also *initializes* a directory, if it hasn't been initialized. Initialization opens a directory for access to its files. The MOVE command - an exception to the rule - doesn't require initialization, but other commands do. Starting up RDOS automatically initializes the master directory, but you must specifically initialize other directories to use them. DIR does this for you.

Now you can check the current directory with another useful command:

```
GDIR)
MACRODIR
R
```

For neatness, let's delete the original files in master directory Dxx. First, get back to the master directory:

```
DIR %MDIR%)
R
```

%MDIR% is a CLI variable that contains the master directory name. You can use it in command lines just as you'd use Dxx.

Now to delete the original files:

```
DELETE/V FIELA.MC FILEB.MC)
FILE DOES NOT EXIST: FIELA.MC
DELETED FILEB.MC
R
```

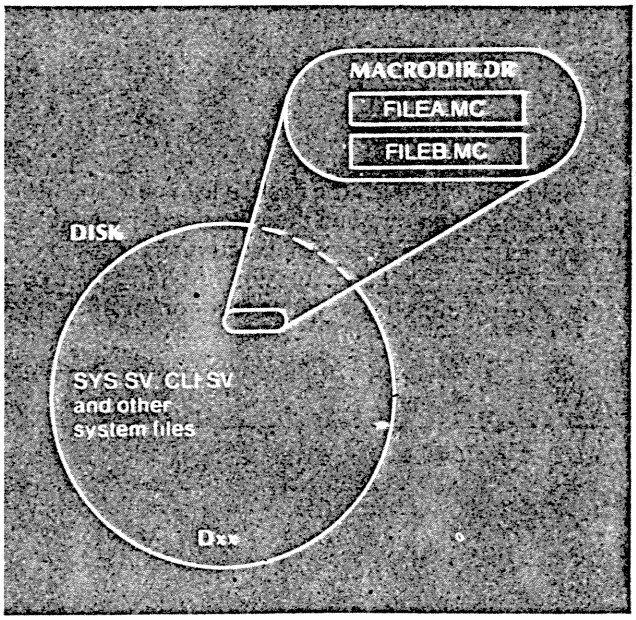
Typos are no-noes.

```
DELETE/V FILEA)
FILE DOES NOT EXIST: FILEA
R
```

(sigh)

```
DELETE/V FILEA.MC)
DELETED FILEA.MC
R
```

At this point, your disk looks like Figure 2-2; the current directory is Dxx.



SD-00730

Figure 2-2. Disk with Subdirectory

Whenever you create a directory, the CLI assigns the extension .DR to its name. The CLI also assigns different extensions to files after they have been assembled, compiled, or loaded. .SV is another CLI-assigned extension. These extensions come in very handy when you want to access specific kinds of files. For example:

```
LIST□-.DR)
MACRODIR.DR 512  DY
R
```

As explained in the Preface, the box represents a space; we use it only where a space isn't obvious in the format. The dash (-) is a *template character*, which in this case instructs the CLI to list *all* filenames that have the .DR extension. (Directories are also files.) Try it with save files:

```
LIST□-.SV)
SYS.SV     39424 SD
DSKED.SV   12288 SD
RLDR.SV    4096  SD
CLI.SV     10240 SD

CTRL-A
INT
R
```

You could also have used the dash to list your .MC files on Dxx. Instead, you moved them and they're probably better off in their own directory. You could also list all files without an extension by typing LIST -.), or files whose names began with a letter or number:

```
LIST□S-.-)
SYS.SV     39424 SD
SEDIT.SV   9216  SD
SYS.OL     27648 C

CTRL-A
INT
R
```

Another template character is the asterisk, which means *one* character, as opposed to the dash, which means from zero to ten characters. To list all five-letter filenames which begin with F and have two-character extensions, you'd type:

```
LIST□F****.**)
FLOAD.SV  15072 SD
FDUMP.SV  15360 SD
R
```

These files are the fast dump and fast load programs in the master directory. Note the absence of FILEA.MC and FILEB.MC, which are in MACRODIR.

Templates work only with certain commands, in certain contexts. Don't be afraid to experiment with them (except with DELETE); at worst, you'll get an error message.

At this point, you can explore disk directories further. Your disk has the master directory (Dxx) and one subdirectory (MACRODIR). Because you've just started using your system, you don't have enough different kinds of files to require another directory. Eventually you will want others, and you may want some of them to be *secondary partitions*. So, make sure Dxx is the current directory, and create a secondary partition:

```
DIR %MDIR%)
R
CPART ALPHA     128)
R
LIST□-.DR)
ALPHA.DR       65536 CTY
MACRODIR.DR    512   DY
R
```

MACRODIR is a subdirectory and works by pointing to other files, hence it will never show more than 512 bytes. ALPHA is a secondary partition, and will physically contain any files and subdirectories you place in it: naturally, these can never exceed the 128 disk blocks (65,536 bytes) you specified for ALPHA (128 blocks at 512 bytes each equals 65,536 bytes). Proceed to build your file hierarchy:

```
DIR ALPHA)
R
CDIR BETADIR)
R
LIST)
BETADIR.DR     512   DY
R
```

Give each new directory a little file:

```
XFER/A $TTI INALPHA.MC)
MESSAGE I'M A FILE IN SECONDARY)
MESSAGE PARTITION ALPHA.)
CTRL-Z
R
DIR BETADIR)
R
XFER/A $TTI INBETADIR.MC)
MESSAGE I'M A FILE IN ALPHA'S)
MESSAGE SUBDIRECTORY, BETADIR)
CTRL-Z
R
```

Now, your disk structure looks like Figure 2-3, the current directory is BETADIR.

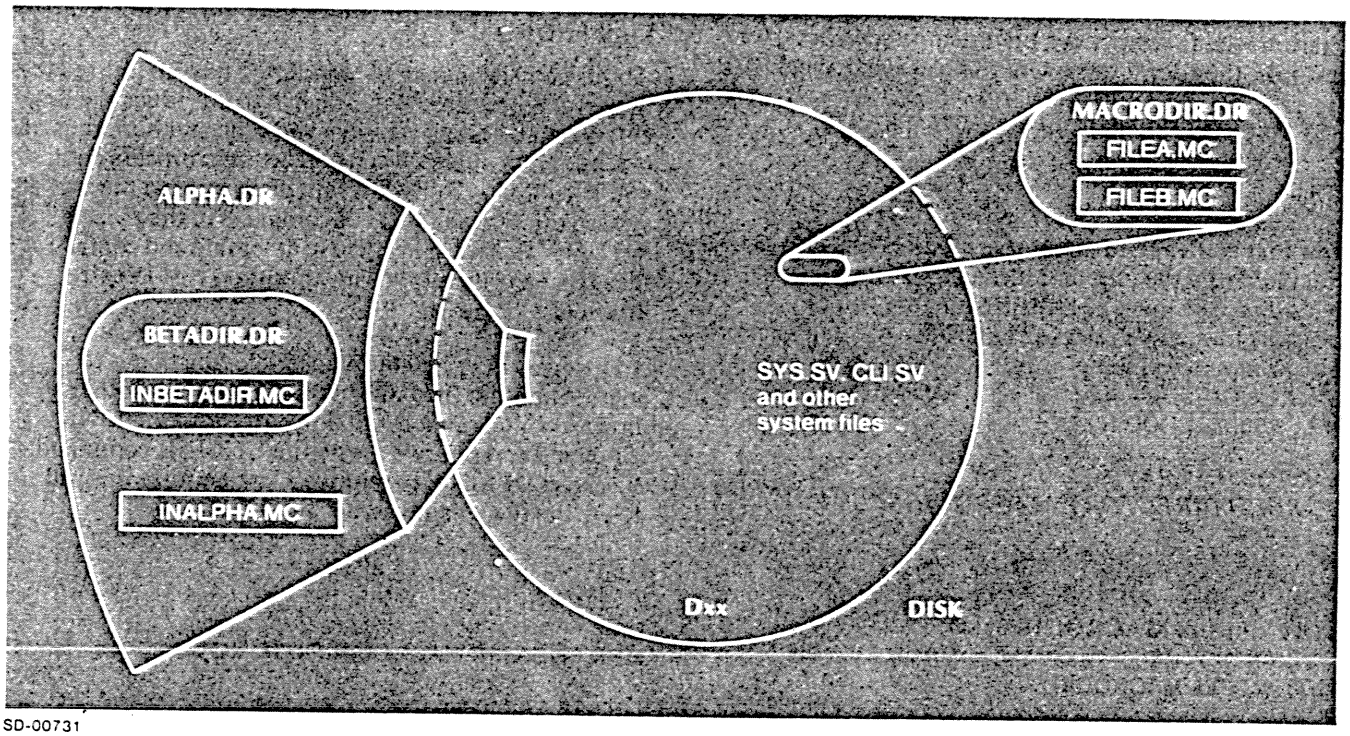


Figure 2-3. Disk with Secondary Partition and Subdirectories

The master directory, Dxx, is the *primary partition;* it has one secondary partition, ALPHA, and one subdirectory, MACRODIR, ALPHA, in turn, has one subdirectory, BETADIR.

Your directory structure is growing. Take a breather, and offer thanks for the GDIR and DIR commands. If the hierarchy seems too complex, you can always delete the new directories later.

The system allows you to execute any file simply by typing the directory name, a colon, and the file name (of course, you could always DIR to the directory you wanted, then type the file name, but that requires an extra step).

```
GDIR)
BETADIR
R
INALPHA)
FILE DOES NOT EXIST: INALPHA.SV
R
```

(True, in directory BETADIR.) Try a directory specifier:

```
ALPHA:INALPHA)
I'M A FILE IN SECONDARY
PARTITION ALPHA.
R
```

Eureka. The directory specifier (:) also works with most other CLI commands:

```
LIST ALPHA:INALPHA.MC)
ALPHA:INALPHA.MC 57
R
TYPE ALPHA:INALPHA.MC)
MESSAGE I'M A FILE IN SECONDARY
MESSAGE PARTITION ALPHA.
R
```

Try it with FILEA.MC in MACRODIR:

```
MACRODIR:FILEA)
HELLO
HOW MANY BLOCKS ARE LEFT
MESSAGE ON THIS DISK?
LEFT:97  USED:31
WHAT'S THE CURRENT DIRECTORY?
BETADIR
R
```

This example shows two things: that DISK always returns the space left in the current *partition* (here, ALPHA), and that GDIR always returns the current directory name (even though a file in another directory issues GDIR).

There are some restrictions on directory specifiers; for example, they don't work with template characters:

```
LIST MACRODIR:-.MC)
R
```

Now that you've tried directory specifiers, you can use them to clarify the idea of initialization. The rules say that a directory must be initialized before you can access its files. Let's release MACRODIR (remove its initialization), and see.

```
RELEASE MACRODIR)
R
MACRODIR:FILEA)
NO SUCH DIRECTORY:MACRODIR:FILEA.SV
R
```

So we initialize MACRODIR again:

```
INIT %MDIR%:MACRODIR)
R
MACRODIR:FILEA)
HELLO
    .
    .
R
```

We had to specify %MDIR% to initialize MACRODIR because MACRODIR is in the master directory, not in the current directory. When you initialize, you can choose between two commands: INIT and DIR. Use INIT when you don't want to change the current directory, and DIR when you do.

There are situations where directory specifiers appear useful, but are not. They don't help when the file specified needs another file and can't find it; for example:

```
MACRODIR:FILEB)
FILE DOES NOT EXIST:FILEA.MC
R
```

Remember FILEB.MC? You wrote it to type FILEA.MC, and it did so; but now it cannot find FILEA.MC in the current directory, BETADIR. There are two ways to handle this: the obvious way, which requires deleting FILEB.MC and recreating it with a directory specifier (MACRODIR:FILEA.MC), and the challenging, interesting way, which involves creating a *link entry* to FILEA.MC.

Link Entries

At this point, you may be wondering how all this relates to useful processing. Well, we had to introduce the link entry somewhere; it can save you time and hundreds of blocks of disk space. When you begin to process data, you'll be linking from different directories to the big system utility assemblers or compilers in the master directory; this will allow you to assemble and compile files from any directory with almost no cost in extra disk space.

A link entry is a filename whose sole function is to point to another filename in the same or in another directory. You can create a link entry with the LINK command and remove it with the UNLINK command; *Don't* use DELETE to remove a link because this will delete the file you linked to.

```
GDIR)
BETADIR
R
LINK FILEA.MC MACRODIR:FILEA.MC)
R
LIST FILEA.MC)
FILEA.MC MACRODIR:FILEA.MC
R
```

This is what a link entry looks like: no byte count or file organization key letter.

Now, when FILEB.MC searches current directory BETADIR for FILEA.MC, the link entry named FILEA.MC will point to MACRODIR:

```
MACRODIR:FILEB)
MESSAGE HELLO
MESSAGE HOW MANY BLOCKS ARE LEFT
MESSAGE ON THIS DISK?
    .
    .
    .
R
```

Your link entry allowed FILEB to find FILEA. If you're dubious, remove the link:

```
UNLINK/V FILEA.MC)
UNLINKED FILEA.MC
R
MACRODIR:FILEB)
FILE DOES NOT EXIST:FILEA.MC
R
```

Convinced? Recreate the link:

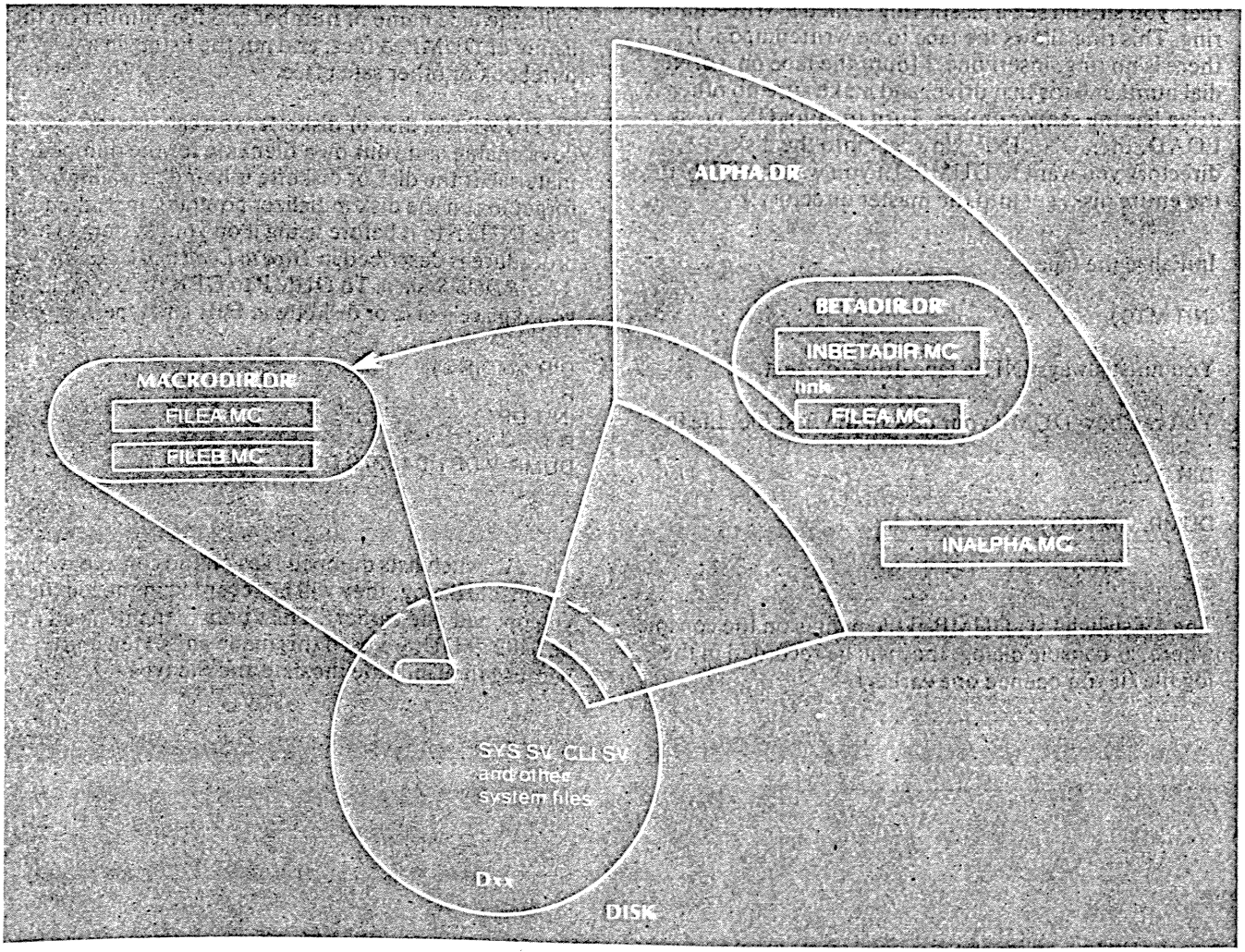
```
LINK FILEA.MC MACRODIR:FILEA.MC)
R
```

You can use any name you want for a link, but if you forget it, your link will be useless. In this example, the link name had to be FILEA.MC because FILEB.MC would search for that name only.

You'll use the same procedure to create all links. For example, the RDOS Macroassembler uses three files, which can require up to 250 disk blocks. You can link to the files from any directory, use the assembler, and consume only a few bytes per link.

You have now created files, a subdirectory, a secondary partition, another subdirectory, and a link entry, and used the commands LOG, CRAND, LIST, DELETE, XFER, TYPE, MESSAGE, DISK, GDIR, RENAME, CTRL-A (not really a command, but an interrupt), CDIR, MOVE, DIR, CPART, INIT, RELEASE, LINK, and UNLINK. You've also used switches and template characters, - and *, and CLI variable %MDIR%.

Your disk structure now looks like Figure 2-4.



SD-00732

Figure 2-4. Disk with Secondary Partition, Subdirectories and a Link Entry

Concluding the Session

Before terminating your session, you should back up your disk material using the DUMP command. DUMP produces a special kind of copy file, which you can then LOAD back onto disk later should something either happen to the disk or if you delete the original files for more disk space. Both DUMP and LOAD offer useful switches, which can select files by date; they also accept template characters.

DUMP can copy the current disk to mag tape, or a disk file. Unless you specify files, it copies the entire contents of the current directory, including all subordinate directories. To use mag tape, take a new reel and examine the inner part of the reel. On a new reel, you should see a plastic ring, called a write-enable ring. This ring allows the tape to be written upon. If there is no ring, insert one. Mount the tape on a drive, dial number 0 for that drive, and make sure no other drive has the same number. Turn the unit ON, press LOAD, then ON LINE. Now, get into the disk directory you want to DUMP. (If you want to DUMP the entire disk, get into the master directory.)

Initialize the tape:

```
INIT MT0)
```

You must always INIT a tape drive to use it.

You can now DUMP your disk to the first file, file 0:

```
DIR Dxx)
R
DUMP/V MT0:0)
    .
    .
```

The /V switch lists DUMPed file names on the console where, as console dialog, they will be recorded in the log file (if you opened one earlier).

DUMP takes a little while to execute as it copies all files and file directory information to file 0 of the tape; eventually you'll get the R prompt back. Type:

```
RELEASE MT0)
R
```

This will rewind the tape and release the drive from the system. Take the drive off line, UNLOAD the tape, remove it from the drive, and store it; turn the drive OFF. Label the tape, noting the contents of the dump, the date, and the tape file number. The file number is very important; if you forget it and inadvertently DUMP to the same file later, you'll lose the original file and all subsequent files on the tape. 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 99. Now, write the tape name or number and file number on the listing of DUMPed files, and put the listing in a notebook or other safe place.

To DUMP to a disk or diskette, you can use the devicename and your own filename for the dumped material. If the disk or diskette is brand new, don't forget to run the disk initializer program on it, then type INIT/F to it before using it on your system. This procedure is described in *How to Load and Generate Your RDOS System.* To DUMP to DP1, for example, you'd place a disk or diskette in DP1 and type:

```
DIR %MDIR%)
R
INIT DP1)
R
DUMP/V DP1:040279.DU)
    .
    .
```

The /V switch lists dumped files on the console, where, as console dialog, they will be recorded in the log file (if you opened one earlier). The dump filename need not be 040279.DU; we used this name and extension because they indicate the date and file type.

Normally, whatever device you're dumping to, you'll want to dump selected files, not the entire disk. To dump files by date, use the /A switch, which instructs the CLI to dump only those files created or modified on or after a specific date. The counterpart of /A (after) is /B (before).

```
DUMP/V {MT0:0          } 04-01-79/A)
       {DP1:040279.DU  }
```

You DUMP any specific directory from within it:

```
DIR ALPHA)
R
DUMP/V {MT0:n          } )
       {DPn:040279.DU  }
```

By dumping a directory, you dump its contents; hence, dumping ALPHA also dumps BETADIR.

You can also use the template characters within *the current directory*. For example, to dump all save and overlay files from current directory Dxx, you would type:

```
DUMP/V destinationfile -.SV -.OL)
```

Later on, to return the DUMPed files to the current directory, you must use the LOAD command. The /R switch, described under LOAD in Chapter 4, conflicts with identical filenames that exist in the current directory. After readying the source device, you must INIT it, then do the LOAD:

```
DIR %MDIR%)
INIT  { MT0 } )
      { DP1 }
R
LOAD/V {MT0:0          } )
       {DP1:040279.DU  }
```

You can also use local date switches and the template characters in the LOAD command.

At this point, we assume that you've DUMPed all the files you created during this session. If you want, you can now delete them and restore the disk to its original state with only the system files on master directory Dxx. This requires little effort:

```
DIR %MDIR%)
R
RELEASE ALPHA)
R
DELETE/V ALPHA.DR)
DELETED ALPHA.DR
R
RELEASE MACRODIR)
R
DELETE/V MACRODIR.DR)
DELETED MACRODIR.DR
R
```

The RELEASE command removes the directory or device that you introduced by INIT (or DIR) to the system. It also releases any subordinate directories. At this point, if you opened log file LOG.CM earlier, you might want to close it and print it. This will give you a hard-copy record of your dialog with the CLI. If you have a line printer, turn it on, place it ON LINE, and type:

```
ENDLOG)
R
DIR %MDIR%)
R
PRINT LOG.CM)
      .
      .
R
```

If you have no line printer, but have a printing console connected to the second keyboard/printer interface, type:

```
ENDLOG)
R
DIR %MDIR%)
R
XFER/A LOG.CM $TT01)
      .
      .
      .
R
```

To access LOG.CM, you must be in, or use a directory specifier to, the master directory because you started the log file in the master directory. For more on LOG and any other commands you've used, see the command in Chapter 4 or in the *CLI User's Manual.* The log file is quite large by now (check it with LIST LOG.CM); you might want to delete it.

Now, you can release the master directory and shut down the system. You can release any directory from any directory -- including itself. Type:

```
RELEASE %MDIR%)
```

Releasing the master directory updates the disk with all new file information and shuts down the operating system. The CLI and RDOS sign off with the message

MASTER DEVICE RELEASED

You can now flip the disk switch to LOAD, wait for the LOAD light (if this applies), then turn off the disk drive, computer, console, and line printer.

Congratulations. You've just completed a session with RDOS. This isn't a toddler's lesson; it includes most of the concepts and commands you'll use in your day-to-day interaction with RDOS, and it provides a sound background for the intricacies of other CLI commands and RDOS itself.

You can find details on certain CLI commands in Chapter 4; these are covered more deeply in the *CLI User's Manual.* You may feel ready to proceed to Chapter 5, which describes using the Superedit text-editing utility.

End of Chapter

Chapter 3
At the Console - DOS

This chapter describes a sample session with DOS and the CLI, and leads you through many of the things you would do on the keyboard to create and organize files. When you've finished the session, you'll have a practical working knowledge of DOS basics. (For a session with RDOS, see Chapter 2.)

If you inadvertently depart from a step we describe, your later experiences may differ from the text description. This is ok; mistakes are a primary vehicle for learning and the CLI will usually prevent disasters. If you get really lost, go back to the beginning of the section you are in and give the files and directories different names; for example, FILEA1.MC instead of FILEA.MC.

Later on, you can find more detail on the features and ramifications of your commands in the *RDOS/DOS CLI User's Manual* or the *DOS Reference Manual*. If you generated your own DOS system, you already have some experience with DOS.

This session assumes that you'll be using a hard disk with diskette or dual-diskette drive and that the system disk(ette) will be in drive 0. You can, however, do almost everything in this chapter with a single diskette drive.

Within this chapter and book, the word *disk* means either hard disk or diskette; *diskette* means only diskette.

Program Load Steps (Bootstrapping)

Turn your system console ON, and make sure it is ON LINE. On some DASHER™ displays, the LINE switch is in back of the console. If this console has both upper- and lowercase letters, set it in uppercase mode with the ALPHA LOCK key because the program that brings up the system doesn't accept lowercase letters. Later you can change back to upper- and lowercase mode.

Turn or press the computer power switch to ON or RUN, whichever applies. Turn the disk or diskette drive ON. If your DOS system is on a removable hard-disk cartridge, make sure this cartridge is in its drive; press the LOAD/READY switch to READY. For any hard disk, wait for the READY lamp to light.

For diskette-based DOS, make sure the write-protect hole of the system diskette is taped. Then insert this diskette in diskette drive 0 and close the door. (If you don't know which drive is 0, try the left; then if program loading doesn't work, try the right. The one that works is drive 0.)

Look at the system console. If it shows an exclamation point (!) prompt, then you have a virtual or *programmed* console. If it shows nothing, then your computer has either a hand-held console, console debug, or CPU program load.

Computers with Programmed Consoles

For a microNOVA machine with a programmed console, find n in Table 3-1. Type nL (for example, 100026L) on the *system* console. Skip to *Bringing Up DOS*.

For a NOVA computer with a programmed console, type 100033L on the system console. Skip to *Bringing Up DOS*.

Table 3-1. microNOVA Device Codes

Disk type:	n =
Hard, sealed disk (hard disk without a LOAD/READY switch).	100026
Hard dual-platter disk subsystem (disk drive with a LOAD/READY or LOAD/RUN switch).	100027
Double-density diskette (these drives have a lamp in the center of the latch).	100026
Single-density diskette (these drives have three lamps above the diskette slot).	33

Computers without Programmed Consoles

If you have a microNOVA computer with a hand-held console (a small box that is initially attached to the computer panel but can be detached; looks like a small calculator), press the RESET and CLR-D keys on the hand-held console. Find the appropriate nin Table 3-1 (for example, 100026) and enter nin the hand-held console with the number keys. Press the PR LOAD key and skip to *Bringing Up DOS.*

If your microNOVA doesn't have a hand-held console, it has either console debug or CPU program load. Press the rocker switch on the computer front panel to the PL/START position. If the system console then displays a question, you have CPU program load; skip to *Bringing Up DOS*, below. If the rocker switch has no effect, then you have console debug. This works the same way as a programmed console, so go back and execute the steps described for programmed console machines.

If you have a NOVA computer with hardware data switches, look at the computer front panel and make sure the data switches are set to 100033 octal (switches 0, 11, 12, 14, and 15 up, the others down). If the switches aren't set properly, fix them. (The data switches are toggle-type switches, numbered 0 through 15.) Lift the RESET switch, then the PROGRAM LOAD switch and proceed.

Bringing Up DOS

After your program load steps, the system console will ask:

FILENAME?

It's asking for the name of your DOS system. (If nothing happens, place the diskette in the other drive, and execute the program load steps again.) If the person who generated this system gave it any name other than SYS, you must type that name; but let's assume it was named SYS. To indicate SYS, simply press the RETURN key ():

FILENAME?)
DOS REV x.xx (*x.xx* is the revision number)
DATE (M/D/Y)?

If you receive the message *FILE NOT FOUND: SYS.SV*, make sure that the diskette in DP0 contains a system, and try to discover (or remember) its name; then type this name and answer the date/time questions:

FILENAME?)
DOS REV x.xx
DATE (M/D/Y)? 4 2 79)
TIME (H:M:S)? 13 10) (hours in 24)
R

R is the CLI prompt; it means that the CLI is ready to accept a command. You'll see it hundreds of times in the future. You can now switch to upper- and lowercase, if this applies. DOS will translate lowercase letters to uppercase.

Typing Mistakes, Control Characters, and BREAK Key

If you type a faulty command and press), the CLI will respond with an error message - usually *FILE DOES NOT EXIST:faulty-entry* - and return the R prompt. If you want to correct a line before pressing return, simply press the DEL or RUBOUT key sequentially to erase characters from right to left. (On printer consoles, DEL or RUBOUT echoes a backarrow (←) or underscore (_) for each character erase.) You can erase an entire line by typing backslash (\) or SHIFT-L.

To stop execution of any CLI command or utility program (except an editor), press the CTRL key, hold it down, then press A. CTRL-A halts the command or program and returns the R prompt.

You can interrupt output to your console at any time by typing CTRL-S. The output won't be lost; you can continue it from the point of CTRL-S by typing CTRL-Q. You can use CTRL-S and CTRL-Q intermittently to read a long file on a display console.

If your computer has a programmed console (!prompt), the BREAK key on the system console keyboard stops execution of all software. To continue with the program that was running, type P) (uppercase P, then)). We mention this here because, in the course of bringing up and running DOS, you may inadvertently hit the BREAK key. If everything stops and the system console shows !, try typing P).

The Session

If your system console has a display screen instead of a printer, you should record console dialog in the disk log file so that you can print and review it later. If the system console has a printer, it provides reviewable copy as you type; you don't need a disk log file and can proceed to the section called *Master Directory Name.*

Log File

With a display terminal, type

```
LIST LOG.CM)
```

If there is an old LOG.CM log file on the disk, the screen will display

```
LOG.CM
```

and a number and letter, then the R prompt. Preserve the old log file by typing

```
RENAME LOG.CM LOGOLD.CM)
R
```

Next, whether or not there was an old log file, type

```
LOG/H)
```

to create a new log file, open it, and start recording console dialog in it. Later, to close the log file, you'll type ENDLOG).

Master Directory Name

Next you should discover your master directory name, so type:

```
MDIR)
```

The CLI returns the master directory name, which varies with the type of disk you have. Generally, it is either DE0, DP0, or DH0. We use *Dxx* to indicate the master directory name, so you should mentally substitute the name returned from MDIR) for *Dxx* in this session.

Creating Some Files

The next step, naturally, is to create a file. The CLI offers several file-creating commands, and we choose CRAND:

```
CRAND FILEA)
R
```

Voila. You now have an empty disk file named FILEA. FILEA was an *argument* to your command; it told the CLI what to name the new file. Some CLI commands require arguments, others don't.

You can verify FILEA's existence with the LIST command:

```
LIST FILEA)
FILEA.   0 D
R
```

and remove FILEA with the DELETE command:

```
DELETE/V FILEA)
DELETED FILEA
R
```

The /V is a *switch,* which modifies the basic meaning of a command; here, it told the CLI to verify the deletion. Now, create a different version of FILEA:

```
CRAND FILEA.MC)
R
```

Now there's an empty FILEA.MC on the disk. Normally, you'd use a text editor to put something in it, but you haven't reached Chapter 5 -- *Creating and Editing Text* -- yet. You can, however, insert text by transferring it directly from the console into FILEA.MC:

```
XFER/A/B $TTI FILEA.MC
MESSAGE HELLO)
CTRL-Z      (Hold down CTRL key and press Z)
R
```

The CTRL and Z keys won't echo as characters on the console. On a display terminal, the R prompt may leap to the top of the screen when you type CTRL-Z. If so, type CTRL-L to clear the screen and proceed with a clear screen.

You've just transferred the CLI command MESSAGE and the text string HELLO into FILEA.MC. The XFER command transfers the contents of one file (the console input file, STTI) to another file (disk file FILEA.MC). XFER requires two arguments; in this case, they were STTI and FILEA.MC. The /A switch specifies ASCII transfer; /B tells XFER to append to the existing file. If you omitted /B, XFER would try to create FILEA.MC, which would produce an error message because FILEA.MC already exists.

Try to execute FILEA.MC:

```
FILEA)
HELLO
R
```

Your FILEA.MC is a CLI macro. The .MC extension tells the CLI to execute all commands within it. You can omit the .MC extension to *execute* a file, but you must include it for all other commands that involve the file. Check FILEA.MC's statistics with LIST:

```
LIST/E FILEA.MC)
FILEA.MC 14 D 04/02/79 13:20 04/02/79 [000444] 0
R
```

The /E switch tells the CLI to list every statistic about the file. These include byte-length (14), organization type (D means random), date and time created or last modified, date last opened, starting disk block address in octal, and use count. Obviously, the central four categories will differ for your own FILEA.MC.

To check the file's contents, type:

```
TYPE FILEA.MC)
MESSAGE HELLO
R
```

Now, you can add a little sophistication to FILEA.MC:

```
XFER/A/B $TTI FILEA.MC)
MESSAGE HOW MANY BLOCKS ARE LEFT)
MESSAGE ON THIS DISK? ;DISK)
MESSAGE WHAT'S THE CURRENT DIRECTORY?)
GDIR)
CTRL-Z
R
```

One problem with inserting text this way is that you can't edit. If you make a mistake and don't correct it before you terminate the line with a), you must delete the file and rebuild it from the beginning. Having made no mistakes, check the improved macro:

```
FILEA)

HELLO
HOW MANY BLOCKS ARE LEFT
ON THIS DISK?
LEFT:2048 USED:408
WHAT'S THE CURRENT DIRECTORY?
Dxx
R
```

This DISK figure is for a double-density diskette. For a hard disk, it will be larger, for a single-density diskette, smaller.

FILEA.MC is much larger now, as you can see:

```
LIST FILEA.MC)
FILEA.MC      118 D
R
```

FILEA.MC now contains four MESSAGE commands, a DISK command, and a GDIR command. Each would work separately, but you've combined them. (Your own DISK figures may differ from those here).

Proceed to create another macro:

```
XFER/A $TTI FILEB)
TYPE FILEA.MC)
CTRL-Z
R
```

Here, FILEB doesn't exist, so XFER will create it, and you can omit the /B switch. By doing so, you saved two steps, the CRAND FILEB and the /B switch, but you've been a little careless:

```
FILEB)
FILE DOES NOT EXIST:FILEB.SV
R
```

The CLI looked for FILEB.MC, then for the save file, FILEB.SV; finding neither, it returned the error message. To fix it, give FILEB the .MC extension:

```
RENAME FILEB FILEB.MC)
R
FILEB)
MESSAGE HELLO
MESSAGE HOW MANY BLOCKS ARE LEFT
MESSAGE ON THIS DISK?;DISK
    .
    .
    .
R
```

You planned to have your second file type your first file and succeeded. Now, you'd like to compare the files, so you type LIST/E without an argument, as in Figure 3-1.

Your command told the CLI to list all nonpermanent files in the master directory; who knows how long the listing would have continued if you hadn't hit the CTRL-A keys to interrupt the command and return the CLI prompt. (Your own listing will differ from that above). The letters following the byte count indicate the file type and organization.

There are some old friends shown in Figure 3-1: SYS.SV, CLI.SV (save files always end in .SV), and FILEA.MC. If you generated your own system, you'll recognize BOOT.SV. Unfortunately, FILEB.MC didn't show up. You could list the entire contents of directory DP0 to find FILEB.MC, but there's a better way: create a directory for the two macro files.

Creating Some Directories

Proceed to create a directory:

```
CDIR MACRODIR)
R
```

Now, to move the macros into the new directory.

```
MOVE/V MACRODIR FILEA.MC FILEB.MC)
FILEA.MC
FILEB.MC
R
```

The MOVE command copies the files into the new directory: the /V switch instructs the CLI to verify their names as they arrive. Now, the files are safely in MACRODIR; you can make MACRODIR the current directory and then list the files in it:

```
DIR MACRODIR)
R
LIST)
FILEA.MC  118
FILEB.MC  14
R
```

The DIR command makes MACRODIR the current directory. DIR also *initializes* a directory, if it hasn't been initialized. Initialization opens a directory for access to its files. The MOVE command - an exception to the rule - doesn't require initialization, but other commands do. Starting up DOS automatically initializes the master directory, but you must specifically initialize other directories to use them. DIR does this for you.

```
LIST/E)
SYS.SV      36864  SD    03/26/79  13:36  03/26/79  [002707]   0
DSKED.SV    18432  SD    03/29/78  16:18  03/29/78  [001670]   0
RLDR.SV      4608  SD    13/23/78  13:16  03/23/78  [015741]   0
CLI.OL      43008  C     10/05/78  10:14  03/30/79  [000366]   1
BOOT.SV      6656  SD    10/05/78  23:03  10/05/78  [000522]   0
FILEA.MC      118  D     04/02/79  13:24  04/02/79  [000444]   0
CLI.SV      10752  SD    10/05/78  10:14  10/05/78  [003467]   0
CTRL-A
INT
R
```

Figure 3-1. Master Directory Listing

Now you can check the current directory with another useful command:

```
GDIR)
MACRODIR
R
```

For neatness, let's delete the original files in master directory Dxx. First, get back to the master directory:

```
DIR %MDIR%)
R
```

%MDIR% is a CLI variable that contains the master directory name. You can use it in command lines just as you'd use DP0.

Now to delete the original files:

```
DELETE/V FIELA.MC FILEB.MC)
FILE DOES NOT EXIST: FIELA.MC
DELETED FILEB.MC
R
```

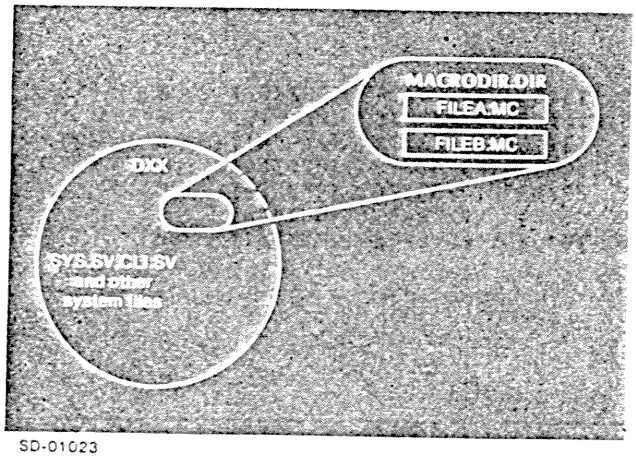
Typos are no-noes.

```
DELETE/V FILEA)
FILE DOES NOT EXIST: FILEA
R
```

(sigh)

```
DELETE/V FILEA.MC)
DELETED FILEA.MC
R
```

At this point, your disk looks like Figure 3-2; the current directory is Dxx.



SD-01023

Figure 3-2. Disk with Directory

Whenever you create a directory, the CLI assigns the extension .DR to its name. The CLI also assigns different extensions to files after they have been assembled, compiled, or loaded. .SV is another CLI-assigned extension. These extensions come in very handy when you want to access specific kinds of files. For example:

```
LIST□-.DR)
MACRODIR.DR   512   DY
R
```

As explained in the Preface, the box represents a space; we use it only where a space isn't obvious in the format. The dash (-) is a *template character*, which in this case instructs the CLI to list *all* filenames that have the .DR extension. (Directories are also files.) Try it with save files:

```
LIST□-.SV)
SYS.SV      39424  SD
DSKED.SV    12288  SD
RLDR.SV     4096   SD
CLI.SV      10240  SD

CTRL-A
INT
R
```

You could also have used the dash to list your .MC files on DP0, but you moved them instead, and they're probably better off in their own directory. You could also list all files without an extension by typing LIST -.), or files whose names began with a letter or number:

```
LIST□S-.-)
SYS.SV      39424  SD
SEDIT.SV    9216   SD
SYS.OL      27648  C

CTRL-A
INT
R
```

Another template character is the asterisk, which means *one* character, as opposed to the dash, which means from zero to ten characters. To list all five-letter filenames which begin with F and have two-character extensions, you'd type:

```
LIST□F****.**)
FLOAD.SV   15072  SD
FDUMP.SV   15360  SR
R
```

These files are the fast load and fast dump programs in the master directory. Note the absence of FILEA and FILEB, which are in MACRODIR.

Templates work only with certain commands, in certain contexts. Don't be afraid to experiment with them (except DELETE); at worst, you'll get an error message.

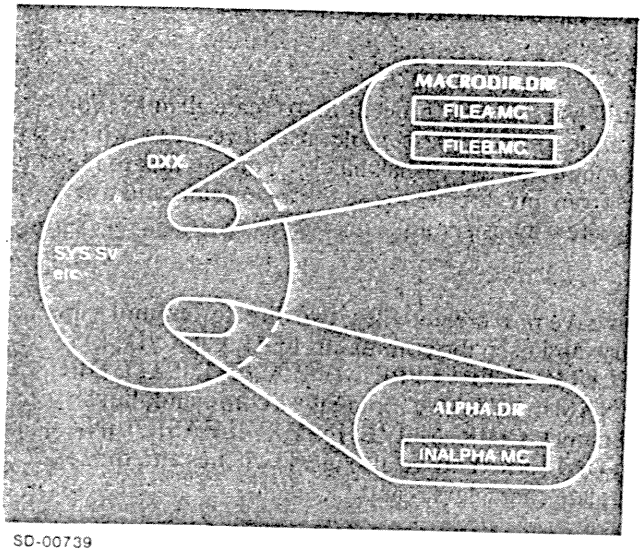
At this point, you can explore disk directories further. Your disk has the master directory (Dxx) and one other directory (MACRODIR). Because you've just started using your system, you don't have enough different kinds of files to require another directory; but eventually you will want others. So, make sure Dxx, is the current directory, and create another directory.

```
DIR %MDIR%)
R
CDIR ALPHA)
R
LIST□-.DR)
MACRODIR.DR  512DY
ALPHA.DR     512DY
R
```

Now, you can proceed with your file hierarchy by giving the new directory a little file:

```
DIR ALPHA)
R
XFER/A $TTI INALPHA.MC)
MESSAGE I'M A FILE IN DIRECTORY ALPHA.)
CTRL-Z
R
DIR %MDIR%)
R
```

Your disk structure looks like Figure 3-3. The current directory is ALPHA.



SD-00739

Figure 3-3. Disk with Two Directories

The system allows you to access or execute any file simply by typing the directory name, a colon, and the file name (of course, you could always DIR to the directory you wanted, then type the file name, but that requires an extra step).

```
GDIR)
Dxx
R
INALPHA)
FILE DOES NOT EXIST: INALPHA.SV
R
```

(True, in directory Dxx.) Try a directory specifier:

```
ALPHA:INALPHA)
I'M A FILE IN DIRECTORY ALPHA.
R
```

Eureka. The directory specifier (:) also works with most other CLI commands:

```
LIST ALPHA:INALPHA.MC)
INALPHA.MC   39
R
TYPE ALPHA:INALPHA.MC)
MESSAGE I'M A FILE IN DIRECTORY ALPHA.
R
```

Try it with FILEA.MC in MACRODIR:

```
MACRODIR:FILEA)
HELLO
HOW MANY BLOCKS ARE LEFT
MESSAGE ON THIS DISK?
LEFT:2022 USED:434
WHAT'S THE CURRENT DIRECTORY?
Dxx
R
```

This example shows that GDIR always returns the current directory name (even though a file in another directory issues GDIR).

There are some restrictions on directory specifiers; for example, they don't work with template characters:

```
LIST MACRODIR:-.MC)
R
```

Now that you've tried directory specifiers, you can use them to clarify the idea of initialization. The rules say that a directory must be initialized before we can access its files. Let's release MACRODIR (remove its initialization), and see.

```
RELEASE MACRODIR)
R
MACRODIR:FILEA)
NO SUCH DIRECTORY: MACRODIR:FILEA
R
```

So we initialize MACRODIR again:

```
INIT MACRODIR)
R
MACRODIR:FILEA)
HELLO
    .
    .
R
```

When you initialize, you can choose between two commands: INIT and DIR. Use INIT when you don't want to change the current directory, and DIR when you do.

There are situations where directory specifiers appear useful, but are not. They don't help when the file specified needs another file and can't find it; for example:

```
MACRODIR:FILEB)
FILE DOES NOT EXIST: FILEA.MC
R
```

Remember FILEB.MC? You wrote it to type FILEA.MC, and it did so; but now it cannot find FILEA.MC in the current directory, DP0. There are two ways to handle this: the obvious way, which requires deleting FILEB.MC and rewriting it with a directory specifier (MACRODIR:FILEA.MC), and the challenging, interesting way, which involves creating a *link entry* to FILEA.MC.

Link Entries

At this point, you may be wondering how all this relates to useful processing. Well, we had to introduce the link entry somewhere; it can save you time and hundreds of blocks of disk space. When you begin to process data, you'll be linking from different directories to the big system utility assemblers or compilers in the master directory; this will allow you to assemble and compile files from any directory with almost no cost in extra disk space.

A link entry is a filename whose sole function is to point to another filename in the same or in another directory. You can create a link entry with the LINK command and remove it with the UNLINK command; *DON'T* use DELETE to remove a link because this will delete the file you linked to.

```
GDIR)
Dxx
R
LINK FILEA.MC MACRODIR:FILEA.MC
R
LIST FILEA.MC)
FILEA.MC MACRODIR:FILEA.MC
R
```

This is what a link entry looks like; no byte count or file organization key letter.

Now, when FILEB.MC searches current directory DP0 for FILEA.MC, the link entry named FILEA.MC will point to MACRODIR:

```
MACRODIR:FILEB)
MESSAGE HELLO
MESSAGE HOW MANY BLOCKS ARE LEFT
MESSAGE ON THIS DISK? :DISK
    .
R
```

Your link entry allowed FILEB to find FILEA. If you're dubious, remove the link:

```
UNLINK/V FILEA.MC)
UNLINKED FILEA.MC
R
MACRODIR:FILEB)
FILE DOES NOT EXIST: FILEA.MC
R
```

Convinced? Recreate the link:

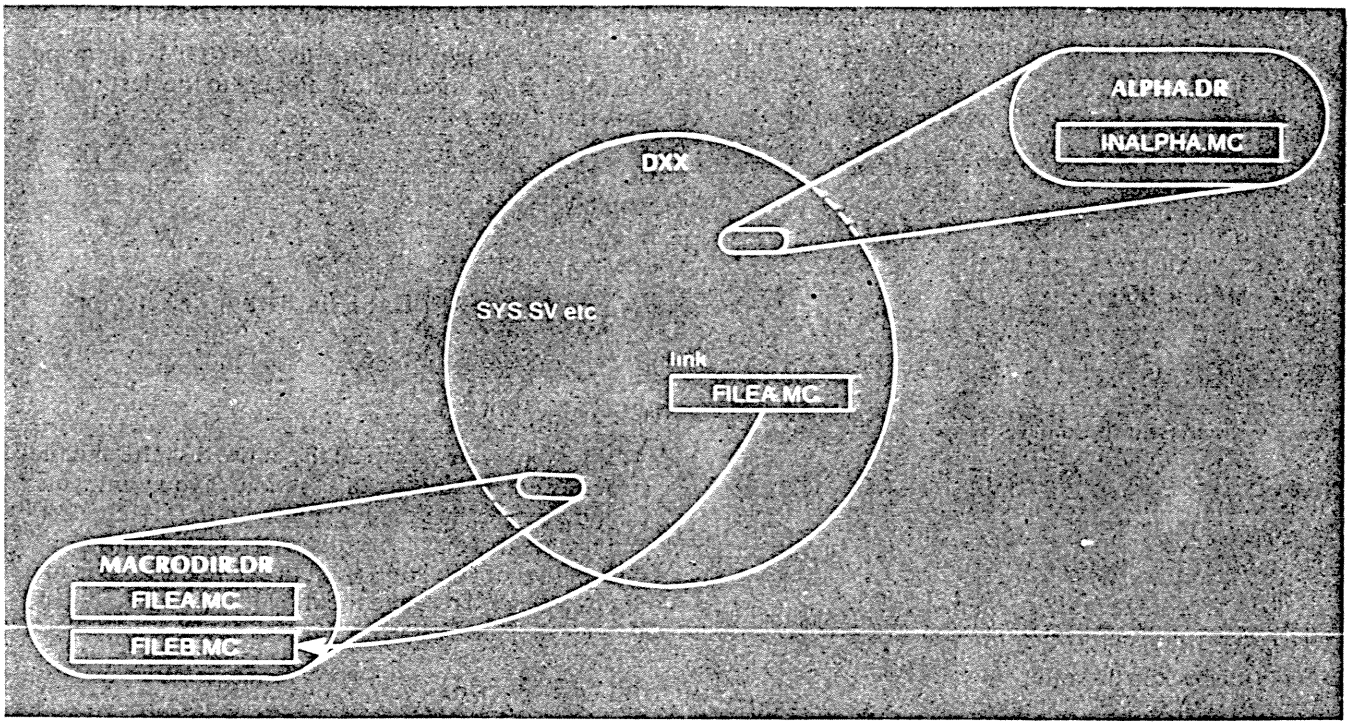
```
LINK FILEA.MC MACRODIR:FILEA.MC)
R
```

You can use any name you want for a link, but if you forget it, your link will be useless. In this example, the link name had to be FILEA.MC because FILEB.MC would search for that name only.

You'll use the same procedure to create all links. For example, the Macroassembler uses three files, which can require up to 250 disk blocks. You can link to the files from any directory, use the assembler, and use only a few bytes per link.

You have now created files, two directories, and a link entry, and used the commands LOG, CRAND, LIST, DELETE, XFER, TYPE, MESSAGE, DISK, GDIR, RENAME, CTRL-A (not really a command, but an interrupt), CDIR, MOVE, INIT, RELEASE, LINK, and UNLINK. You've also used switches and the template characters, - and *.

Your disk structure now looks like Figure 3-4.



SD-00740

Figure 3-4. Disk with Directories and a Link Entry

Concluding the Session

Before terminating the session, you might want to back up your disk material on a diskette. Take a fresh diskette, tape the write-protect hole, and insert it in the appropriate diskette drive. This drive is DE1 if "Dxx" is DE0; otherwise it is DP1. If this copy diskette hasn't been initialized with DOSINIT, read the appropriate Chapter (7 or 8) of *How to Generate Your DOS System, Backing Up Your Files.*

The dump procedure itself depends whether the master directory Dxx is (1) a double-density diskette or hard disk or (2) a single-density diskette.

File Backup on a Double-density Diskette or Hard-disk System

On this kind of system, you'll create facsimile backup directories on your backup diskette and MOVE the files into them. To do ALPHA, type the following commands:

```
INIT DE1)     (for NOVA systems, substitute DP1 for
R              DE1 in this section)
CDIR DE1:ALPHABU)
R
INIT DE1:ALPHABU)
R
DIR ALPHA)
R
MOVE/V ALPHABU)
```

(names of files copied to directory ALPHABU)

```
R
```

ALPHADMP now contains copies of all nonpermanent files in ALPHA. Follow the same procedure for MACRODIR:

```
CDIR DE1:MACROBU)
R
INIT DE1:MACROBU)
R
DIR MACRODIR)
R
MOVE/V MACROBU)
     .
     .
R
```

Now, do the master directory:

```
DIR Dxx)
R
MOVE/V DE0)
     .
     .
R
```

Release the backup diskette:

```
RELEASE DE1)
```

In the future, perhaps daily, you can follow the same MOVE procedure for these and other directories you create under DOS. But in the future, you can use the /R switch in the MOVE command. Normally, the CLI won't MOVE a file if its name already exists in the destination directory. But if you include the /R switch, the CLI will check both files' creation dates and, if the file you want to move is newer, the CLI will delete the file in the destination directory and replace it with the newer version. The CLI will take no action on the destination file if the destination file is newer.

You can couple the /R switch with an /A date switch for the move each day to ensure that the dump directory has the current versions of all files. For example, assume that (as shown in this chapter) the date is April 2, 1979. At the end of tomorrow's session, to back up ALPHA's files, you'd type:

```
MOVE/V/R ALPHABU 4-3-79/A)
     .
R
```

The CLI will then move any ALPHA files that had been created or changed on April 3rd into ALPHABU. If no files in ALPHA had been created or modified on April 3rd, no files will be copied.

File Backup On Single-Density Diskettes

For a single-density diskette, use the COPY command, which copies every directory and file to the destination diskette. Type:

```
COPY DP0 DP1)
(pause)
R
```

(Never INIT or RELEASE the backup diskette for COPY.)

Cleaning Up

At this point, you've copied the entire diskette and file structure created during this session. If you want, you can now delete the new directories on the system disk and restore this disk to its original state with only the system files. If you don't want to delete these, skip to *Shutting Down*. To delete them, type:

```
RELEASE ALPHA)
R
DELETE/V ALPHA.DR)
 DELETED ALPHA.DR
RELEASE MACRODIR)
R
DELETE/V MACRODIR.DR)
 DELETED MACRODIR.DR
R
```

Using the Backup Diskette

If something happens to one or more of the files or directories on your original master disk, you can simply insert the backup in DE1 or DP1 and DIR to it. To replace a directory on the original, DELETE the original directory, recreate it with CDIR, INIT it, then get into the appropriate backup directory and MOVE all files to the newly-created directory in the original. To replace a file, simply DIR to the backup directory and MOVE the file to the original directory.

To make the diskette a true backup from which you can bootstrap a DOS system, you should install a bootstrap root on it via the BOOT BOOT command, as detailed in the appropriate After System Generation chapter in *How to Generate Your DOS System*. This will allow you to bring up DOS on the backup and rebuild the original disk if the original's file structure is badly damaged.

Shutting Down

At this point, if you opened log file LOG.CM earlier, you might want to close it and print it. This will give you a hard-copy record of your dialog with the CLI. If you have a printing console connected to the second teletypewriter interface, type:

```
ENDLOG)
R
XFER/A LOG.CM STTO1)
     .
     .
R
```

If you have a line printer, place it ON LINE and type:

```
ENDLOG)
R
PRINT LOG.CM)
     .
     .
R
```

You don't need a directory specifier to access LOG.CM (although a specifier would do no harm) because you started the log file in the current directory. For more on LOG and any other commands you've used, see the command in the *CLI User's Manual*. The log file is quite large by now (check it with LIST LOG.CM); you might want to delete it.

Now you can RELEASE the system. The RELEASE command removes a directory or device that INIT or DIR introduced to the system. In so doing, it updates the directory on disk. You can RELEASE any directory from any directory -- including itself. To shut down the system, RELEASE the master directory:

```
RELEASE %MDIR%)    (or Dxx)
```

The CLI and DOS will shut down; their final message will be:

MASTER DEVICE RELEASED

You can now remove all diskettes from their drives and turn off the disk drives, computer, console and line printer (if any). Don't forget to write a label for the backup diskette (e.g., DOS SYSTEM BACKUP and the date) and apply the label. Never write on a diskette label after applying it -- unless you use a felt-tipped pen. After labeling, insert each diskette in its outer envelope and store it safely.

Congratulations. You've just completed a session with DOS. This isn't a toddler's lesson; it includes most of the concepts and commands you'll use in your day-to-day interaction with DOS, and it provides a sound background for the intricacies of other CLI commands, DOS itself, and the next chapters.

You can find details on certain CLI commands in Chapter 4; these are covered more deeply in the *CLI User's Manual*. You may feel ready to proceed to Chapter 5, which describes using the Superedit text-editing utility.

End of Chapter

Chapter 4
Commom CLI Commands

The CLI is a powerful system utility that provides many commands, plus a macro facility so you can create still more. As you saw in the last chapter, you can do a lot of work using just a small subset of CLI commands.

In this chapter, we will describe CLI commands in greater detail. Most of these commands have features that are not described here. For a complete description of every command, see the *RDOS/DOS CLI User's Manual.*

From the sample session, you have some sense of *switches.* We divide switches into two categories. Those that affect entire commands are called *global* switches; those that affect one argument in the command are called *local* switches. For example, say you want to copy all files with the extension .SR to directory XDIR *except* files whose names begin with MYPROG, and you want console verification. You would type

```
MOVE/V□XDIR□-.SR□MYPROG-/N)
```

/V is the global switch which instructs the CLI to verify files moved. /N is a local switch which directs the CLI not to move files beginning with MYPROG.

Wherever you can use the template characters (- and *) in a command, we have noted this.

If the CLI command you want to type is too long to fit on a single line, simply type an uparrow (SHIFT-6 or SHIFT-N) immediately before you type). (Note that ↑ works only in CLI commands. It does not function this way in other programs.) You can now continue the command on the next line. Naturally, you omit the ↑ before the last) in the command line. For example,

```
CRAND□↑)
XFILE)
R
```

You can also stack CLI commands on one line by typing a semicolon between them; e.g.,

```
CRAND MYFILE;LIST MYFILE)
MYFILE  0  D
R
```

BUILD

Build a file of filenames

Format:

BUILD newfilename filename *[filename...]*

Build a new file (i.e., a list) of filenames in the current directory. The files need not exist, unless you use the template characters as part of a *filename*. A *filename* can include a directory specifier if it doesn't include a template character. The original filenames will remain intact.

Global Switches:

/A Include all filenames that have permanent and nonpermanent attributes (see CHATR command in the CLI manual to set attributes).

/K Do not include link entries.

/N Do not include extensions to filenames in newfilename.

Local Switches:

mm-dd-yy/A Include only the names of files created this date or after. Arguments mm (month) and dd (day) can be one or two digits. Date switches include *all* matching links unless you include the global /K switch.

mm-dd-yy/B Include only the names of files created before this date.

name/N Do not include names that match name in newfilename.

Template Characters

Permitted

Examples:

```
BUILD ABC-.SR TEST-.)
```

This command creates file ABC and writes two categories of filenames into it: those with the .SR extension, and those which begin with TEST and have no extension.

```
BUILD□MYTEXT□-.TX□A-.-/N)
```

This command creates MYTEXT and inserts the names of all files with the .TX extensions except filenames beginning with A into it.

```
BUILD OUTPUT□-TEXT)
R
PRINT□@OUTPUT@)
      .
      .
      .
```

This sequence produces file OUTPUT, which contains all filenames whose last four characters are TEXT and have no extensions; it then prints these files on the line printer. You can use BUILD to create a file group for input to *any* commands that don't allow a template, like PRINT above. (The sign instructs the CLI to use the *contents* of the file; it is further described in the CLI manual.)

CDIR

Create a subdirectory (RDOS) or directory (DOS)

Format:

CDIR (sub)directoryname

The new (sub)directory will receive the .DR extension.

Switches:

None

Examples:

```
DIR DP1)
R
CDIR BETH)
```

DIR makes DP1 the current directory; CDIR creates (sub)directory BETH.DR on DP1. CDIR DP1:BETH is an equivalent command.

CPART

Create a secondary partition (RDOS)

Format:

CPART partname blockcount

Create the secondary partition named partname with the length specified in blockcount. A disk block is 512 bytes or 256 words. The new partition will be a contiguous disk file and will receive the extension .DR.

You cannot create a partition with less than 48 disk blocks. Also, if blockcount is not a multiple of 16, RDOS truncates it to the nearest lower multiple.

Switches:

None

Examples:

```
DIR DP4F)
R
CPART ALEPH 128)
R
```

The DIR command makes primary partition DP4F the current directory; CPART creates secondary partition ALEPH on DP4F. An equivalent command would be CPART DP4F:ALEPH 128). ALEPH is 128 disk blocks long and is logically distinct from primary partition DP4F.

CRAND

Create a randomly organized file

Format:

CRAND filename *[filename ...]*

Each random file receives the D (random file) attribute and a length of zero; it will grow as required during use.

Switches:

None

Examples:

CRAND RANDFILE)

This command creates RANDFILE in the current directory.

CRAND ACCUFILE DP1:GROWTHFILE)

This command creates ACCUFILE in the current directory and GROWTHFILE in DP1.

DELETE

Delete one or more files or directories

Format:

DELETE filename *[...filename]*

Delete the files in a directory having names given in the argument list. To remove a link entry, you must use the UNLINK command. To delete a directory, RELEASE the directory, then type DELETE directoryname.DR.

Global Switches:

/C Confirm each deletion. Each filename is repeated while the system waits for you to confirm the deletion by typing a carriage return. To prevent the deletion, press any key other than return.

/L List deleted files on SLPT (overrides /V).

/V List names of deleted files on the console.

Local Switches:

mm-dd-yy/A Delete only files created this date or after. Arguments mm (month) and dd (day) can be one or two digits.

mm-dd-yy/B Delete only files created before this date.

name/N Do not delete any files that match this name.

Template Characters:

Permitted only when filename argument is in the current directory.

Examples:

DELETE/V LIMIT.-)

This command deletes all files which have the name LIMIT and any extension (including none); e.g., LIMIT.SR, LIMIT.RB, LIMIT.SV, and LIMIT. This also verifies their names on the console.

DELETE/V□A**B)

This command deletes all files with four-character filenames that begin with A, end with B, and have no extensions. It also verifies files deleted.

DELETE/V□A-B)

This command deletes all files whose names begin with A, end with B, and have no extensions.

DELETE/C□-.LS)

A.LS:) * Delete file A.LS
COM.LS:) * Delete COM.LS
MAP.LS: Don't delete MAP.LS

Confirm before deleting. The system asks for a confirmation of each deletion. When you confirm a deletion with a carriage return, the system echoes an asterisk (*). Any other character echoes a carriage return.

DIR

Change the current directory

Format:

DIR directory *[:subordinate directory]*

At bootstrap time, the master directory becomes the current directory. The DIR command specifies another device or directory as the current device or directory. If necessary, this command will also initialize the device or directory.

Switches:

None

Examples:

DIR ACCTSDUE)

This command makes directory ACCTSDUE, on the current partition or disk, the current directory.

DIR DP1:DEF)

This command makes directory DEF, on DP1, the current directory.

DISK

List the number of disk blocks used and remaining

Format:

DISK

This command returns the decimal number of blocks used and the number of blocks left on the current partition or diskette. If the current directory is a (sub)directory, DISK reports on its parent partition.

Switches:

None

Examples:

```
DISK)

LEFT:2048  USED:400
```

The response indicates that 2048 out of a total of 2448 blocks are still available for use. The figure could pertain to a secondary partition or a double-density diskette.

DUMP

Copy one or more files to an output file

Format:

DUMP outputfilename *[filename...]*

Use this command to copy files from the current directory onto a given file or device outputfilename. If you specify *filename* (with or without template), only matching files are dumped. If you omit *filename*, all nonpermanent files in the current directory are dumped; this includes subordinate directories, if any.

Use LOAD to restore DUMPed files to the current directory.

Global Switches:

/A Dump all files, permanent and nonpermanent.

/K Do not dump links.

/L List the dumped files on SLPT (overrides /V).

/V Verify DUMPed filenames on the console.

Local Switches:

mm-dd-yy/A Dump only files created this date or after. Arguments mm (month) and dd (day) can be one or two digits. Date switches dump *all* matching links unless you include global /K switch.

mm-dd-yy/B Dump only files created before this date.

name/N Don't dump files that match name.

Template Characters:

Permitted only when *filename* argument is in the current directory.

Examples:

```
DUMP/A/L MT0:0 2-20-79/A)
```

This dumps all permanent and nonpermanent disk files created on or after February 20, 1979 onto file 0 (the first file) of magnetic tape unit number 0. It also sends a listing to the line printer. You can then save this tape file as a backup for the disk.

```
DUMP/A/L DP4:SOURCE -.SR 7-14-77/A)
```

This tells the system to dump all files with the .SR extension created on or after July 14, 1977 to file SOURCE on DP4, and to send a listing of filenames to the printer.

```
DUMP/V□MT0:0 -.-□-.RB/N□TEMP.DR
```

Dump all nonpermanent files in the current directory (except .RB files) and directory TEMP and all its files, to file 0 of the tape MT0.

ENDLOG

Close the LOG file

Format:

ENDLOG *[password]*

Close the log file which you opened by a previous LOG command. You must close this file before you can TYPE, PRINT or DELETE it. If the previous LOG command included a *password* argument, you must use the password with the ENDLOG command.

This command, ENDLOG *password*, appears in the log file.

Switches:

None

Examples:

```
ENDLOG GSTONE)
```

The password GSTONE is used since it was specified when the log file was last opened.

Note that you must type the full name of the log file to PRINT or DELETE it; this name is LOG.CM.

GDIR

Display the current directory name

Format:

GDIR

Switches:

None

Examples:

GDIR)
MANHATTAN

MANHATTAN is the current directory.

INIT

Initialize a directory or tape drive

Format:

INIT {tape drive / directory *[:subordinate directory]*}

By initializing a device or directory, you introduce it to the operating system; before the introduction, all of its files are inaccessible.

After you INIT a directory or tape drive you can access all files within it; you can use the colon specifier (e.g., MYDIR:FILEA) at will. All files on the initialized tape unit, directory, or diskette are available until you release (RELEASE command) the device or directory.

Global Switches:

/F Full initialization of a tape drive rewinds the tape and writes two EOF's at its beginning, effectively erasing the tape. On a disk, INIT/F builds a new file directory and free storage map, effectively destroying all existing files. Always think twice before typing INIT/F to anything.

Local Switches:

None

Examples:

INIT DP4

This command initializes the disk in DP4.

INIT MT1)

This command initializes tape drive MT1 and enables access to the tape unit.

INIT DP1:ABC)

All files in directories DP1 and ABC will now be accessible, provided you direct file access with explicit directory specifiers.

LINK

Create a link to a file in another directory

Format:

LINK { resfilename/2
linkentryname *[directory specifier:]* resfilename }

resfilename is the file you're linking to. If your link entry will have the same name as the resolution file (resfilename), and resfilename is in the current directory's parent directory, you can create your link with the first command format (the local /2 switch specifies the same filename). Linkentryname will always be created in the current directory unless you specify another directory.

If the resolution file is not in the current directory's parent, or if your link name will differ from the resolution file's, use the second command format. The resolution file resfilename may exist in any initialized directory.

Global Switches:

None

Local Switches:

linkentryname/2

Create linkentryname to a resolution file of the same name in linkentryname's parent partition. The parent partition is the disk or secondary partition that contains the current directory.

Examples:

```
MDIR)
DE0
R
DIR MYDIR)
R
LINK EDIT.SV/2)
```

The master directory is DE0, and it contains (sub)directory MYDIR. DE0 is MYDIR's parent directory, so the LINK EDIT.SV/2 links to EDIT.SV (the Text Editor) in directory DE0. Typing LINK EDIT.SV/2 is easier than typing LINK EDIT.SV DE0:EDIT.SV, which would have the same effect.

```
DIR DP1)
LINK NSPEED.SV DP0:NSPEED.SV)
LINK SPEED.ER DP0:SPEED.ER)
```

This command creates two link entries named NSPEED.SV and SPEED.ER in DP1 to the editor files on DP0. This permits anyone in directory DP1 to use the NOVA (NSPEED) Supereditor.

```
LINK ASM.SV DZ0:ASM.SV)
```

This command creates a link entry named ASM.SV in the current directory to the extended assembler in directory DZ0.

```
DIR FORT4)
LINK FORT.SV DP0:FORT.SV)
LINK FIV.SV DP0:FIV.SV)
LINK ASM.SV DP0:ASM.SV)
```

These commands create link entries to the FORTRAN IV compiler files and to the extended assembler (which the compiler needs) in directory FORT4.

LIST

List file directory information

Format:

LIST *[directory:][filename...]*

For each file, listed information includes the filename, size, and organization type (except for link entries). You can specify other data with global switches.

If you omit an argument, the system lists all nonpermanent files and link entries in the current directory.

The link access attributes, if any, are preceded by a slash. For link entries, the list includes the link name, the resolution file name, and the directory specifier (if any) given when the link was created. A commercial at sign (@) indicates that the resolution file is on the parent partition or diskette. The following link directory lists common file attributes (or link access attributes) and their meanings:

Letter	Meaning
P	Permanent file; cannot be deleted or renamed.
S	Save file, i.e.; executable program.
W	Write-protected file; cannot be modified.
A	Attribute-protected file whose attributes cannot be changed. The A attribute cannot be removed.

The following is a list of file characteristics and their meanings:

Letter	Meaning
⊐	Space, sequential file organization.
D	Random file organization.
C	Contiguous file organization.
T	This file is a partition (RDOS only).
Y	This file is a directory.

Global Switches:

/A List all files within the current directory, both permanent and nonpermanent.

/C List the creation time.

/E List every category of file information.

/L Output listing on line printer.

/K Do not list links.

/N List links only.

/O List date file last opened (month/day/year).

/S Sort the output listing alphabetically by filename.

Local Switches:

mm-dd-yy/A List files created this date or after. Arguments mm (month) and dd (day) can be one or two digits. Date switches list *all* matching links unless you include the global /K switch.

mm-dd-yy/B List files created before this date.

name/N Do not list files that match this name.

Template Characters:

Permitted only when the *filename* argument is in the current directory.

LIST (continued)

Examples:

LIST/E/A)

This command lists every type of information on all files and link entries in the current directory. A typical line of information would look like this:

FLI.SV 8160 SD 03/23/79 13:56 03/23/79 [000164]0

In this example, FLI.SV is the filename; it consists of 8,160 bytes, is a randomly organized save file, was created (or modified) at 1:56 p.m. on the 23rd day of March 1979, was last opened on that same date, has a starting logical block address of 164_8, and has a file use count of zero.

Typical lines describing link entries would look like this:

ASM.SV DP0:ASM.SV

The link entry name is ASM.SV; the link was created to resolution file ASM.SV on DP0.

EDIT.SV @:EDIT.SV

In this example, the link entry name is EDIT.SV and the resolution file was defined to have the same name and to reside on the parent partition or diskette.

LIST/K/S□-.SV□5-2-79/A)

List all nonpermanent save files (.SV extension) created after May 1, 1979 and sort the list alphabetically. This will not list links and output will go to the console.

LIST/A□-TEXT-.SR□-TEXT1-.-/N)

List all files whose names include the letters TEXT, with the extension .SR, except those filenames which include TEXT1.

LOAD

Load dumped files

Format:

LOAD *inputfilename [filename...]*

Load a previously-dumped file from *inputfilename* int the current directory. If you omit filenames and switches, all nonpermanent files in the input file are loaded. With global switches, you can select filenames for LOADing, or you can choose simply to list on the console or printer the filenames in the input file.

The LOAD command can load only those files that were previously DUMPed. Files you want to load must bear different names from files in the current directory (unless you specify the /N, /O or /R switches).

Global Switches:

/A Load all files, including permanent files.

/K Do not load link entries.

/L List loaded filenames on the line printer (Overrides /V switch and listing by /N.)

/N Do not load files; output the filenames to the console.

/O Delete current file if it exists and replace with file being loaded that has the same name.

/R Select most recent version. If a file in the current directory has the same name as a file in the inputfile, check both files' creation dates. If the version in the inputfile is newer, delete the version in the current directory and replace it with the newer one in the inputfile. If the version in the inputfile is not newer, take no action.

/V Verify the load by listing filenames loaded on the console. Filenames in a directory are listed before the directory name.

Local Switches:

mm-dd-yy/A Load only files created this date or after. Arguments mm and dd can be one or two digits. Date switches load *all* links unless you include the global /K switch.

mm-dd-yy/B Load only files created before this date.

name/N Don't load files that match name.

Template Characters:

Permitted when the file sought is not part of a directory in the inputfile. For example, assume that MYFILE.SR and MYFILE.SV were in directory SUBDIR on DP0. If the DUMP sequence had been DIR SUBDIR; DUMP MT0:0), then LOAD MT0:0 MYFILE-.-) would work. But if the dump sequence had been DUMP MT0:0), then the MYFILE template wouldn't work.

Examples:

```
LOAD MT0:1)
```

This command loads onto disk all previously dumped nonpermanent files on file 1 of the tape on MT0. Filename, length, and attributes are entered in the current directory's file directory.

```
LOAD/L□DP4:DUMP0307-.SV□3-12-79/A□TEST-.-/N)
```

This loads from file DP4:DUMP0307 all files with the .SV extension (except files whose names begin with the characters TEST and files created before March 12, 1979), and lists them on the line printer.

LOG

Start recording in the log file

Format:

LOG *[password][directory/O]*

The log file records all CLI dialog that appears on the console in a file named LOG.CM in the current directory. You cannot examine, print, or delete the log file while it is open. You can close it with the ENDLOG command. Releasing the master device or giving the BOOT command also closes it. LOG.CM records dialog cumulatively; after anyone types LOG, it will append new dialog to the old. To start a new log file in the current directory, RENAME or DELETE the old one, then type LOG).

You can use a *password* to prevent the log file from being closed inadvertently. The password is an optional argument, up to 10 alphanumeric characters in length. If you specify *password*, the same password must be used in the ENDLOG command to close LOG.CM.

The *directory* argument indicates a destination for the log file other than the current directory. You must initialize the directory before using its name.

Global Switch:

/H Place a header, containing directory and date information, at the beginning of the log file.

Local Switches:

directory/O Output the log file to *directory*.

Examples:

```
LOG/H GSTONE)
```

This command outputs all CLI dialog to the file LOG.CM in the current directory. The password is GSTONE.

MOVE

Copy files to any directory

Format:

MOVE destination-directoryname *[filename...]* ↑)
[old filename/S new filename]...

This command will copy a given file or files in the current directory to another directory. *Filename* cannot be a directory. If you omit filenames and switches, all nonpermanent files in the current directory are moved.

Global Switches:

/A Move all files, including permanent files.

/D Delete original files after MOVE.

/K Do not move links.

/L List moved filenames on the line printer.

/R Select most recent version. If a file in the destination-directory has the same name as a file to be MOVEd, check both files' creation dates. If the version to be MOVEd is newer, delete the version in the destination-directory and replace it with the newer version. If the version to be MOVEd is not newer, take no action.

/V Verify MOVEd filenames on the console.

Local Switches:

mm-dd-yy/A — Move any file created or modified this date or after. Arguments mm (month) and dd (day) may be one or two digits. Date switches move *all* matching links unless you include the global /K switch.

mm-dd-yy/B — Move any file created or modified before this date.

name/N — Do not move files that match name.

oldname/S newname — Assign newname to the preceding file but retain its oldname in the current directory.

Template Characters:

Permitted.

Examples:

```
MOVE/D/K□MYDIR□-.SR)
```

This command moves all nonpermanent files in the current directory with .SR extension (except link entries) into destination-directory MYDIR, and deletes the original files after the transfer.

```
MOVE/A ACCTSDUE□-.-□3/1/79/B)
```

This command moves to directory ACCTSDUE all files created or modified before March 1, 1979.

```
DIR MYDIR)
MOVE/V %GDIR% FILEA/S FILEA1)
FILEA1
```

This command copies a file, under a different name, in the current directory. By doing this when you plan extensive changes to a file, you preserve a backup version. In this case, FILEA1 is a backup version of FILEA.

PRINT

Print a file on the line printer

Format:

PRINT *filename [...,filename]*

The source files you wish to print may come from any device.

Switches:

None

Examples:

PRINT FOO.SR DP4:COM.SR EXT.SR)

This command prints source files FOO.SR, COM.SR in DP4, and EXT.SR on the line printer.

PRINT ACCTSDUE:JONES.CO)

This prints file JONES.CO in directory ACCTSDUE.

RELEASE

Release a directory or tape drive from the system

Format:

RELEASE {directory / tape drive}

RELEASEing a directory releases all subordinate directories. When you RELEASE a mag tape unit, the tape is automatically rewound. After release, a directory or device is closed to access until you initialize it with an INIT or DIR command. Be sure to release a disk before physically removing it. By releasing a directory, you release all subordinate directories in it; a release of the master directory releases everything in the system, including tape transports.

Switches:

None

Examples:

RELEASE DP1)
R

This command releases DP1 and all its directories. DP1 was not the master device.

RELEASE DP0F)
MASTER DEVICE RELEASED

This releases DP0F, which was the master device.

RENAME

Rename a file

Format:

RENAME oldname newname *[oldname newname]...*

Rename a file in the current directory. You can rename any nonpermanent file that is not open, but note that some system utilities (e.g., the assemblers or the FORTRAN compiler) will not work if you RENAME them. No save (program) file can execute without the .SV extension.

Switches:

None

Examples:

```
RENAME LOG.CM LOGOLD.CM)
R
```

Rename the current log file LOGOLD.CM. The log file must have been closed via ENDLOG before you can RENAME it.

```
RENAME FILEB FILEB.MC)
R
```

Give FILEB the .MC extension so that the CLI can execute it as a macro file.

TYPE

Type a file on the system console

Format:

TYPE filename *filename...*

Copy an ASCII file or files on the console. The source files may come from any device.

Switches:

None

Examples:

```
TYPE A.SR B.SR DP1:XX.SR)
.
.
.
```

This command displays or types the following disk files on the program console: A.SR and B.SR in the current directory, and source file XX.SR on DP1.

UNLINK

Remove a link entry

Format:

UNLINK linkname *[...linkname]*

Unlink files by removing one or more link entrynames. This does not affect the resolution file.

Global Switches:

/C Confirm each removal. The system displays each link name on the console, and waits for you to confirm the deletion by typing a carriage return. Type any key other than return to retain the link.

/L List the removed links on the line printer (overrides /V).

/V Verify each link entry removed.

Template Characters:

Permitted only when the linkname argument is in the current directory.

Examples:

UNLINK/C□TEST.-)

This command asks for a confirmation before each link entry is removed:

TEST.SR)* Link TEST.SR is removed
TEST.RB)* Link TEST.RB is removed
TEST.SV% Link TEST.SV is not removed

XFER

Copy the contents of a file into another file

Format:

XFER *sourcefile destinationfile*

XFER can transfer a file on any device to another file on any other device. If *destinationfile* does not exist, XFER creates it. XFER differs from MOVE and DUMP in that it transfers *only* the file contents, not the name, creation date, or other file directory information. MOVE and DUMP *do* copy this information. Thus, you cannot XFER a file that has been DUMPed; you must LOAD it first. To copy a binary file (.RB or .SV extension), use the MOVE command. XFER will work for such files only if you execute certain steps afterward as described in the CLI manual, XFER command.

You can XFER console input directly to a file, as shown in Chapters 2 and 3.

Global Switches:

By default, files are transferred without alteration. There are two global switches:

/A ASCII transfer. Transfer the file line by line taking appropriate read/write action, such as inserting line feeds after each carriage return when transfer is from disk to line printer.

/B Append the source file to the end of the destination file. (Required to append to an existing file.)

Examples:

XFER $PTR Q)

This command transfers the file in the paper tape reader to a disk file named Q.

XFER DP0:MYFILE DP1:MYFILE)

This copies the contents of MYFILE in directory DP0 under the name MYFILE file in DP1.

See Chapter 2 or 3 or the CLI manual for other examples.

The Next Steps

If you'll be working exclusively in BASIC, skip all the way to Chapter 7; you don't need the next two chapters.

To code in another high-level language, you'll use a text editor program (described in Chapter 5), a compiler, then another utility to make the compiled code executable. Chapter 6 covers the compiling and processing steps for a FORTRAN IV program; steps for other languages are similar. For the precise details, of course, see the language reference manual for your compiler.

If you're an assembly language programmer, you'll follow the same procedure as the FORTRAN programmer, except that you'll use an assembler instead of a compiler. You'll need Superedit (Chapter 5), then the assembly language information in Chapters 8 and 9.

End of Chapter

Chapter 5
Creating and Editing Text

You received two different editor utilities with your system: the text editor (EDIT) and the Supereditor. Superedit, the topic of this chapter, is the more advanced editor and is much handier than the XFER/A command you used in the last chapter. Within *this* chapter, we explain enough Supereditor features to let you use it. A complete description of the editor is outside the scope of this book; you can find a complete description in the *Superedit Text Editor User's Manual (RDOS/DOS).*

It's sound practice, especially for a new user, to save a backup copy of each text file under a different name.

Superedit Features

The editor is a utility program which you evoke by a CLI command, but its commands bear no relation to the CLI's. Superedit lets you create and modify files containing upper- and lowercase ASCII text. During editing you can change, delete, search for, or insert single text characters, lines of text, or large portions of whole files. The editor is *string-oriented.* This means that its commands work with character sequences, which need not be complete lines. A line of text is a string of characters terminated by a carriage return. You can change or search for character combinations without knowing where they are.

Superedit maintains a *character pointer* (CP), which indicates the current editing position in the file. It also provides a command to show you where the CP is; but for the most part, you simply keep a mental note of the pointer's position. You change the CP's position by executing edit commands. 2L, for example, moves the CP down to the start of the second line beyond its current position.

executing edit commands. 2L, for example, moves the CP down to the start of the second line beyond its current position.

You can enter two kinds of input when you edit: editor commands and text you want in your file. When you are inserting new text, your Superedit command, which may include many lines, will include text you want inserted.

When it is ready to execute a command, Superedit types an exclamation point (!) prompt character. After you see this prompt, you can type in one or more editing commands. You can enter two or more commands on one line by typing an ESC character between each command; you terminate the entire command line by typing ESC twice. When you type ESC, Superedit echoes a dollar sign on your screen; obviously, ESC ESC appears as two consecutive dollar signs. Superedit executes the commands in a multiple command entry sequentially, from left to right. If you enter an incorrect command anywhere in a multiple command line, Superedit informs you of the error, then ignores the remainder of the command line; that is, it processes only those commands to the left of the invalid command.

Note: Although Superedit accepts both upper- and lowercase characters, the FORTRAN compiler and assemblers do not, except in text or comment strings. Thus, for all text you intend to compile or assemble, you must enter letters in uppercase, except for comment or text strings.

Superedit Command Summary

The editor commands we will explain are:

Command	Operation
C	Change a string of text to another string.
H	Halt the Superedit program and return to the CLI.
I	Insert new text.
J	Jump to the beginning of the edit buffer.
K	Kill (delete) one or more lines of text.
L	Move to the beginning of a line.
M	Move n number of characters.
S	Search for a string of text.
T	Type one or more lines.
UE	Close the file, include all updates and write it to disk.
US	Close the file, including all updates, and write it to disk; save the original file under the original filename with the .BU extension.

The *edit buffer* is an area of the computer's memory where Superedit works on your file during the editing process. The commands you type change the buffer; the edited file isn't written to disk until you type UE.

Invoking the Editor

To begin an editing session, get into the directory that holds Superedit. This is often the master directory, but you can check in any directory. DIR to it and type:

```
LIST□-SPEED.SV)
```

After you have found it, type:

```
NSPEED filename )
```

on a NOVA computer; or

```
SPEED filename)
```

on an ECLIPSE computer.

filename is the name of the file you want to edit. If *filename* already exists, Superedit will copy it into the edit buffer. If filename does not exist, Superedit will create a file for you, and you'll begin editing with an empty edit buffer. In the latter case, Superedit will give you the message:

CREATING NEW FILE

If you want to use Superedit from a directory that does not hold its files, DIR to the directory and type:

```
LINK    {SPEED.SV destdir:SPEED.SV   }
        {NSPEED.SV destdir:NSPEED.SV }  )

LINK     SPEED.ER destdir:SPEED.ER)
```

The destdir is the name of the directory that holds the Superedit files -- often the master directory, Dxx or %MDIR%.

To illustrate each Superedit command, we'll create and edit a FORTRAN source program that calculates mortgage payments. We begin by entering the CLI command that executes the editor program:

```
{NSPEED} MORTGAGE.FR)
{SPEED }
```

At this point, Superedit replies:

CREATING NEW FILE
!

The exclamation point is the Superedit prompt; it means that we can start keying in commands. We received the message CREATING NEW FILE because MORTGAGE.FR did not exist before we gave the command.

Typing Mistakes

Often, as you are typing a Superedit command, and before you type ESC ESC, you'll notice that you made a mistake. Superedit gives you three easy ways to correct any mistake:

- DEL or RUBOUT
- CTRL-X
- CTRL-A

Strike the DEL or RUBOUT key to backspace and delete the previous character. Superedit will echo each character rubbed out. You can do this repeatedly to delete some or all of your command to the beginning of the current line. (DEL or RUBOUT will not delete the prompt character.)

If you want to delete the entire current line, simply type CTRL-X. This is equivalent to typing DEL or RUBOUT back to the beginning of the line. CTRL-X deletes the last line only.

Finally, if you want to cancel a command that Superedit is executing (after you've type ESC ESC), type CTRL-A. With the commands we've described here, however, you will usually not be quick enough to stop the whole command from being executed.

A Note of Caution

Superedit is a powerful editor -- practically a text-processing language -- but its power and speed can sometimes make life difficult for a novice user. Until you learn it well, you should update your file often, saving a backup version with the US and H commands (USHS) to minimize lost effort. If text seems to have vanished from the edit buffer after a command, type US$HS to update the file and save the original version, then examine both the current and original versions with either Superedit or the CLI TYPE command and work with the one you want. The US and H commands are detailed later in this chapter.

Insert New Text (I)

When you evoke Superedit, it reads some or all of your file into its buffer (if the file already exists), or simply starts with a clear buffer. In either case, its CP (character pointer) always points to the first position in the buffer. Since you have just created a file, there is nothing to edit and you can start by inserting lines of source code.

```
!tabREAL INT, ITD, LB)
5tabTYPE "ENTER AMOUNT, RATER, YEARS")
tabTYPE "AND, 0 FOR SUMMARY OR 1 FOR")
$$
```

tab represents pressing the CTRL and I keys to produce a tab.

After each insertion, the CP points after the last inserted character. This lets you repeat insert commands in the same order that you would type words or lines of text on a typewriter. In the text above, we could have typed the text between quotes in lowercase, however, the compiler requires the REAL statement in uppercase.)

Generally, you should not insert more than ten lines of text in one I command. It's sound practice to end each Insert with $$ after typing several lines, then continue inserting with a new I command.

Jump CP to the Beginning of the Buffer (J)

At any time in the editing process you can move the CP to the start of the buffer; simply key in the J command. You can also use the L command to move the CP from one line to another, but J gets you to the start of the buffer immediately.

Examine Some Lines in Your File (T)

It's pretty good practice to review each addition or change after you make it, so after every change you should type the T command. There are three variations of this command, and none of them moves the CP:

O,nT	Type the buffer from the beginning to character n. To type the entire buffer, simply insert a number sign before T; i.e., # T.
nT	Starting at the current CP, type the next or previous n lines of the buffer. n is positive (forward) unless you precede it with a minus sign.
-mTnT	Type text from m lines backward to n lines forward, to show the text surrounding the CP.
T	Type the current line and show where the CP is. Superedit uses the 3-character combination (↑) to show where the CP is.

The command

```
2T$$
```

types the current line and the next line, while

```
-2T$$
```

types the two previous lines. In the example above, the string J$3T$$ displays the three lines typed.

Search for a Character String (S)

One of the most useful Superedit commands is the S command. This command lets you find character combinations or strings. A *string* is a sequence of any number of characters (including lowercase letters and invisible characters like)). When you use the S command, you move the CP forward through your file to the character position immediately after some character string. Thus:

```
J$STYPE□
```

moves to the start of the buffer and searches for the string "TYPE □". If we added the T command to the command string:

```
J$STYPE□$T$$
```

Superedit would type the string and show the CP:

```
5TYPE (↑) "ENTER AMOUNT,RATER, YEARS"
!
```

In a search or change command, you must type precisely the string you want. If, for example, you carelessly insert two spaces in the search command, Superedit will not find the string and will display an error message.

```
!J$STYPE□□$T$$
ERROR:UNSUCCESSFUL SEARCH
STYPE □□ ST
```

When a command fails, Superedit types it back after the error message.

You'd get the same message if the CP was beyond the text string you wanted and you specified a search without repositioning the CP:

```
!JS REAL$T$$
REAL (↑) INT, ITD, LB
!SREAL$T$$
ERROR: UNSUCCESSFUL SEARCH
SREALSTSS
!
```

When the editor cannot find a string, it positions the CP at the start of the buffer.

Change Text in Your File (C)

When you key in a C command, you specify the string you want to change and the new string that you want to take its place. Separate these two arguments with an ESC character. The command resembles S in that it searches for a string to change; but C changes characters - thus you must be careful to specify the exact combination of characters you want to change, or you may end up changing a string that you wanted to leave intact. You can also use the Cstring$$ command to delete a string.

When you inserted text with the I command, you typed RATER instead of RATE. Suppose you back up the CP to the beginning of the buffer (using the J command) and tell Superedit to change TER to TE:

```
!J$CTER$TE$T$$
```

(Start at beginning of buffer; change the first occurrence of "TER" to "TE", type line.)

then you end up with a source line that reads

```
5TYPE"ENTE AMOUNT,RATER, YEARS"
```

So, to correct the mistake and verify the correction, type:

```
!JCENTE□$ENTER$T$$
5 TYPE "ENTER (↑) AMOUNT, RATER, YEARS"
!
```

Then fix RATER, being more specific about the change:

```
!CRATER$RATE$T$$
5 TYPE "ENTER AMOUNT, RATE (↑), YEARS"
!
```

This changes RATER to RATE, the change you wanted to make.

You could have done this a different way, by making sure the CP was positioned properly with the S command:

```
!J$STER$T$$
5 TYPE "ENTER (↑) AMOUNT,RATER, YEARS"
!CTER$TE$T$$
5 TYPE "ENTER AMOUNT,RATE (↑), YEARS"
!
```

Set the CP at the Start of a New Line (L)

Often you may want to set the CP several lines forward or backward from its current position. You use the L command to do this, and it has two variations:

L Set the CP to the beginning of the current line.

nL Set the CP to the start of a different line. If n is positive, the CP moves n lines forward from the current line. If you precede it with a minus sign, the CP moves n lines backward.

L moves the CP to the start of the current line, 2L moves the CP to the start of the second line down from the current line.

Again, you can use the T command to check the CP position:

```
!J$T$$
(↑) REAL INT, ITD, LB
!L$T$2L$T$$
(↑) REAL INT, ITD, LB
(↑) TYPE "AND, 0 FOR SUMMARY OR 1 FOR"
!
```

The first command line starts at the beginning of the buffer and types the first line. The second command line moves the CP to the beginning of the line, types the line, then moves the CP two lines forward and types that line.

Move the CP (M)

The M command, format nM, moves the CP backwards or forwards by n number of characters. To move the CP to the left, make n negative; e.g., -2M. To move it to the right, use a positive number; e.g., 2M. Generally, you'll use M to move the CP within a line, but you need not do so. For example, if you move the CP past a carriage return character, it will move into another line of text.

If the current line (displayed by T$$) is:

(↑) *REAL INT, ITD, LB*

And you type the command 2M$T$$

```
!2M$T$$
RE(↑)AL INT, ITD, LB
!
```

Then, type -2M$T$$ to restore the CP to its original position:

```
!-2M$T$$
(↑) REAL INT, ITD, LB
!
```

Delete Lines (K)

Sometimes you may want to delete an entire line. (Often it's easier to delete a bad line and insert a new one than to try to correct the original.) To delete one or more lines, use the K command. This command takes an argument n indicating how many lines to delete:

```
3K$$
```

This command kills (deletes) three lines from the current CP position (the first line is everything to the right of the CP on the current line). 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).

For example, assume you want to delete the second line, which begins with 5. First, use the S, L, and T commands to check the surrounding lines:

```
!J$S5$-1L$3T$$
REAL INT, ITD, LB
5 TYPE "ENTER AMOUNT, RATE, YEARS"
TYPE "AND, 0 FOR SUMMARY OR 1 FOR"
!
```

Then get to the target line with S, and verify the line with T. Get to the beginning of the line with L, delete the line with K, and verify the deletion with T.

```
!S5$T$$
5 (↑) TYPE "ENTER AMOUNT, RATE, YEARS"
!L$1K$-1L$2T$$
REAL INT, ITD, LB
TYPE "AND, 0 FOR SUMMARY AND 1 FOR"
!
```

To restore the line, set CP position *after* the carriage return character in line one, and use the I command:

```
!J$SLB)
$$
!I5tabTYPE "ENTER AMOUNT,RATE,YEARS")
$$
```

Terminate Editing (UE or US)

After you finish editing, you must key in the UE command. This applies all the editing changes you made to this file, and it completes the creation of the new, updated file. To save the original file as a backup file, key in US instead of UE. Superedit will then update the file and save the original file under filename.BU, deleting any older filename.BU first. You can then issue the H command to return to the CLI.

Return to the CLI (H)

The H command returns you to the CLI. You will usually issue this command after UE or US, but you may issue H at any time. You might key in H without UE if you realized that you have made some substantial editing errors, and that it would be easier to start all over again than to try to correct the errors by working on the updated file. H without UE returns you to the CLI immediately, without writing out the edit buffer or copying the remainder of the input file. It leaves your original file intact on disk.

Any command string starting with an H will terminate Superedit and return to the CLI, and all work done in the edit buffer will be lost. Therefore, avoid starting a command string with H accidentally. For instance, you might forget to begin this string with I, S, or C:

```
!HELP RETRIEVE MY EDIT BUFFER$$
```

and Superedit would terminate, returning you to the CLI:

```
R
```

Before turning to the remaining chapters, you should sit down at a terminal and practice with Superedit. Try to create and make some minor modifications to files using Superedit. When you become familiar with Superedit, you can then proceed to learn how to write a program under RDOS or DOS. Via the CLI, you may want to create directories for your Superedit files, LIST, MOVE, or DUMP them.

Summary of Editing Commands

Table 5-1 summarizes each of the commands we've discussed and gives some simple examples of their use.

Table 5-1. Superedit Command Examples

Command	Examples	Result	Command	Examples	Result
#T	#TSS	Type the entire buffer on the console.	M	ZMSTS-ZMSTSS	Move the CP two characters right, type line, then move the CP two characters left, type line.
	3TSS	Type three lines from CP including the current line.			
	TSS	Type the current line and show where the character pointer is.	S	SOUNT,SS	Set the CP after the comma following OUNT as shown in this example string:
	SOUNT, STSS	Given this string.			...ENTER AMOUNT,...
		...ENTER AMOUNT, RATE AS...	C	SOUNT,SS CTERSTESS	Set the CP after OUNT, then change TER to TE Applied to this string,
		this command shows the pointer located immediately after the comma after AMOUNT:			...ENTER AMOUNT, RATER AS...
		...ENTER AMOUNT,(↑) RATE AS...			it produces this result:
					...ENTER AMOUNT, RATE AS...
J	JS5T	Go to the start of the buffer and type the first 5 lines.	I	LS I HELLOSS	Insert the string HELLO before the current line.
L	LS I)) SS	Insert two blank lines before the current line.	K	LS1KSS	Delete the entire current line.
	3LSS	Set the CP to the start of the 3rd line down from the current line.	UE US	UEHSS USHSS	Apply editing changes to the file, close it, and return to the CLI.

End of Chapter

Chapter 6
Instant FORTRAN IV Programming

This chapter assumes that you have the optional FORTRAN IV compiler utility programs, and that you have loaded them, along with the FORTRAN libraries, into your master directory, as described in an appendix of the *FORTRAN IV User's Manual*. The FORTRAN IV compiler consists of two files: FORT.SV and FIV.SV. (If your machine has only 16K of memory, do not use the FIV.SV file; instead use FIVNS8.SV, DUMP or COPY FIV.SV to save it, then delete the original and RENAME FIVNS8.SV to FIV.SV.) The compiler also expects that the Extended Assembler, ASM.SV, is in this directory.

Program Steps

These are the steps you follow to write a FORTRAN IV program.

1) Create or edit a FORTRAN IV source file with Superedit.

2) Compile and assemble the source file with the CLI command

   FORT filename)

   This produces a relocatable binary file.

3) If there are any compile-time errors, go to 1; if not, go to 4.

4) Make the relocatable binary file into an executable save file with RLDR:

   RLDR filename *[subroutine names...]* FORT0.LB↑)
   FORT1.LB FORT2.LB FORT3.LB SMPYD.LB)

5) Run the save file with the CLI command

   filename)

6) If the program is correct, go to 9; if not, go to 7.

7) Diagnose your program using runtime error messages or erroneous output.

8) Go to 1.

9) You're done!

This chapter guides you through all the steps you need to write and execute such a program.

Writing the FORTRAN Source Program

The FORTRAN example in this chapter is a simple program to calculate home mortgage payments; it produces a schedule of monthly principal and interest. The program uses only ordinary arithmetic operations, calls no subroutines, and (excluding comments) is only about half a page long. The program uses two formulas. You need not understand how these formulas work to understand the illustration. We have chosen FORTRAN IV for this program (instead of FORTRAN 5) because FORTRAN IV runs under both RDOS and DOS.

Figure 6-1 shows a flowchart for this program, and Figure 6-2 shows our initial version of the program, MORTGAGE.FR, complete with errors. (We used portions of this program for editing examples in the last chapter.)

We assume that you'll be writing, compiling, and loading this program in the master directory. Later, you might want to create a directory (e.g., FORT4) to hold your FORTRAN IV programs. In this directory, you'd need links to the Superedit files ASM.SV, RLDR.SV, RLDR.OL, SYS.LB, and the FORTRAN libraries (FORTn.LB).

Program MORTGAGE.FR reads input from the console (ACCEPT statement), and writes its results to the console (device number 10). If your system has a line printer, you can direct output to the printer by substituting the printer device number (12) for console output number (10) in each WRITE statement after line 110 (except for statement 40). For example, the line following the format statement in line 110 would become

```
WRITE (12,120) PAY.
```

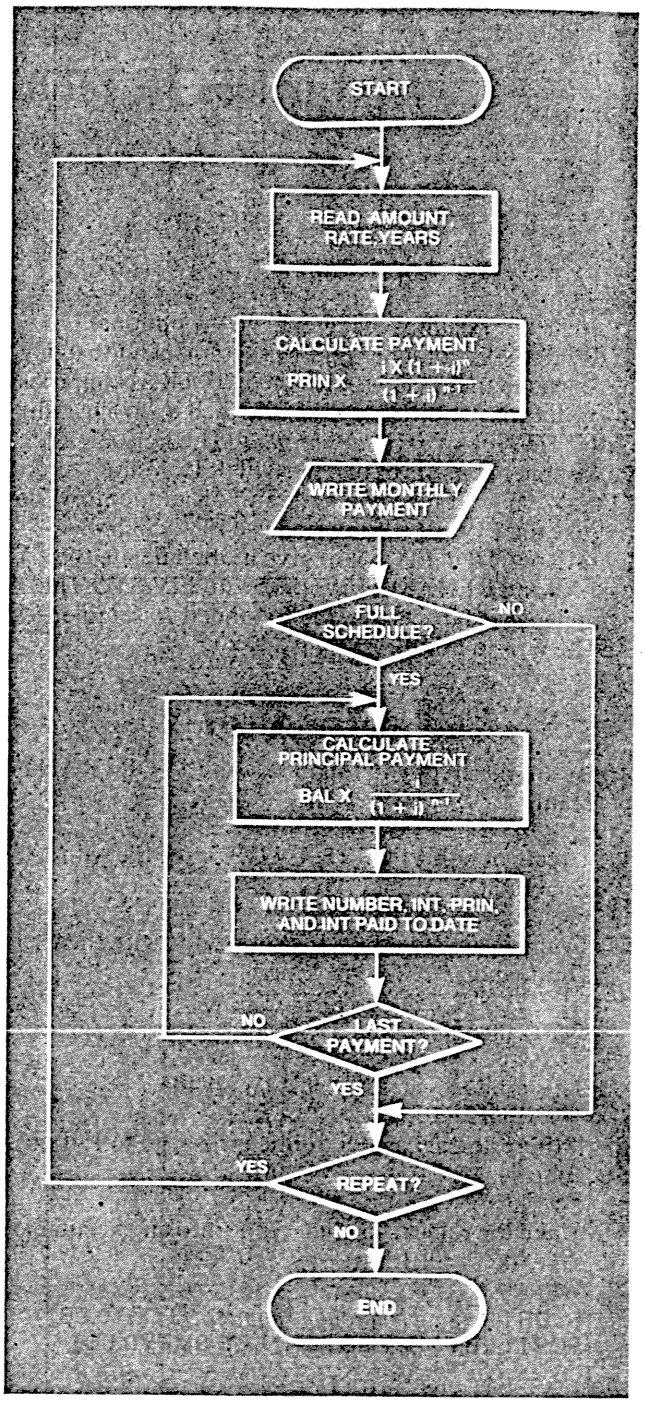
You should examine Figure 6-2 before proceeding to the next section.

Compiling the FORTRAN Program

Having written an initial version of MORTGAGE.FR, we now compile it. The FORTRAN IV compiler produces an intermediate assembly-language program, then invokes the Extended Assembler, which produces a binary program.

Since the initial version probably has some syntactical errors in it, we use the global /N switch, which tells the compiler to scan the input for errors, but produces no output file (producing one would do no harm, but it would take longer). After making sure that the current directory has the compiler files and the Extended Assembler (or links to these), type:

```
FORT/N MORTGAGE)
```



SD-00741

Figure 6-1. MORTGAGE.FR Program Flowchart

```
        REAL INT, ITD, LB               ; Else these would be integers.
5       TYPE "ENTER AMOUNT, RATE, YEARS"
        TYPE "AND, 0 FOR SUMMARY OR 1 FOR"
        TYPE "FULL SCHEDULE.  SEPARATE EACH"
        TYPE "ENTRY WITH A COMMA."
        TYPE "TERMINATE INPUT WITH RETURN."
        ACCEPT  AMOUNT,RATE,IYEARS,IFULL        ; Get figures from console.

C  Change yearly rate RATE to monthly rate R.
        R = RATE/12
C  Change years IYEARS to months N.
        N = IYEARS*12

C  Calculate monthly payment PAY, write heacer and PAY.
        PAY = AMOUNT*R*(1+R)**N/((1+R)**N-1)
        WRITE (10,110) AMOUNT,RATE,IYEARS
110     FORMAT (1H0,"AMOUNT    = $",F9.2,/," INTEREST RATE =",F7.4,/,
     X  " LOAN LIFE IS",I4," YEARS",/)    ; Char. in pos. 6 continues line.
        WRITE (10,120) PAY
120     FORMAT (1H0,"MONTHLY PAYMENT = $",F10.2,/)

C   Summary or full schedule?
        IF (IFULL .LE. 0) GO TO 40

C   Full schedule -- set up variables.
        LB = AMOUNT     ; Initial Loan Balance equals original amount.
        NN = N          ; Save original number of months in NN.
        ITD = 0         ; Interest To Date is initially 0.

C   Write header, then calculate and write figures month by month.
        WRITE (10,130)
130     FORMAT (1H0," NUM",7X,"INTEREST",5X,"PRIN. PAY        PRIN."
     X  "BAL",6X,"INTEREST PAID TO DATE",/)

        DO 30 I=1,NN    ; DO until you reach NN months --
C       Calculate amount of principal in payment, PN.
        PN = LB*R/((R+1)**N-1)
C       Decrement month.
        N = N-1
C       Calculate amount of interest in payment, INT.
        INT = PAY-PN
C       Update loan balance.
        LB = LB-PN
C       Update interest paid to date.
        ITD = ITD+INT
C       Write figures for this month.
        WRITE (10,140) I, INT, PN, LB, ITD
140     FORMAT (1H0,I3,7X,"$",F9.2,"     $",F9.2,"     $",F9.2, 8X,"$", F0.2)
30      CONTINUE

40      ACCEPT " TYPE 1 TO REPEAT, 0 TO STOP. ', ISTOP
        IF (ISTOP .GT. 0) GO TO 5
        END
```

Figure 6-2. MORTGAGE.FR Program With Errors

During the compilation, five errors appear on the screen along with the source lines where these errors may have occurred:

```
;       REAL INT, ITD, LB
; *** 060 *** CHR 10
;       INT = PAY-PN
; *** 003 *** CHR 05
;       WRITE (10,140) I, INT, PN, LB, ITD
; *** 047 *** CHR 01
;       WRITE (10,140) I, INT, PN, LB, ITD
; *** 060 *** CHR 23
;       IF (ISTOP .GT. 0) GO TO 5
; *** 111 *** CHR 01

PROGRAM IS RELOCATABLE

                        .TITL   .MAIN
```

Figure 6-3. Compile-Time Error Messages

The last line isn't an error; it results from the assembler's scan of the program.

Note we said *may have occurred.* Actually the offending line of source code may be located one or two lines before the one displayed with an error code. Looking in Appendix B of the *FORTRAN IV User's Manual,* you'll find two sets of numeric error codes. The first set, entitled *FORTRAN IV ERROR MESSAGES,* are those codes which apply to compile-time errors; the second set, *FORTRAN IV RUN-TIME ERROR MESSAGES,* describes errors you can receive when you execute your program.

The codes shown in Figure 6-3 have the following meanings:

060 Error in formal syntactical structure; identifier is of wrong variety.

003 Unknown statement type.

047 Illegal operand types for current operator.

111 Hollerith constant not ended at statement end.

You shall see that most of these errors result from a single error and, in general, that these error codes do little more than point us in the right direction.

The first error message claims that the error occurs at approximately the 13th character (CHR) position in the line.

```
REAL INT, ITD, LB
```

(Character positions are counted from the left margin, position 1.) The statement's syntax seems ok; but the name of one of our variables, INT, is the same as that of a predefined FORTRAN library function--INT for integer. This caused the problem, and changing the name of this variable to INTEREST clears it up. Because this error caused the compiler to reject the line, you might expect some other lines using the same variables (INT, ITD, and LB) to also be flagged as bad. You would be right; all of the remaining errors except the last result from this first error.

The last error is kind of perplexing because the IF statement syntax is perfect. The error actually occurred on the *preceding* line, statement 40, where we began the Hollerith string with a quote (") and ended it with an apostrophe ('). This often happens in FORTRAN: the compiler flags an innocent line when the preceding line is bad. It occurs because the compiler has already processed the innocent line when it detects the error.

Using Superedit, correct the errors with selective S and C commands; e.g., CINTSINTERESTSTSS. The five erroneous lines should look like this after you change them:

```
REAL INTEREST, ITD, LB
INTEREST = PAY-PN
ITD = ITD + INTEREST
WRITE (10,140) NP, INTEREST, PN, LB, ITD
40 ACCEPT "TYPE 1 TO REPEAT, 0 TO STOP.", ISTOP
```

After these changes, repeat the FORT command, this time omitting the /N global switch. The compilation and assembly should proceed without error messages. When assembly is completed, you'll receive the R prompt, indicating that you can now begin the next step in your procedure: processing with the relocatable loader utility, RLDR.

Creating the Save File

The RLDR command line for MORTGAGE is:

```
RLDR MORTGAGE FORT0.LB FORT1.LB↑)
 FORT2.LB FORT3.LB SMPYD.LB)
```

The -.LB names following MORTGAGE are the names of libraries that you must always use when loading a FORTRAN IV program. Typing the four library names is a nuisance; eventually you may want to merge the libraries under a single name (e.g., FORT.LB) with the Library File Editor (LFE) utility.

Later, for more on the LFE command, see the appropriate appendix of the *FORTRAN IV User's Manual*. Note that if your computer has hardware multiply-divide, you might want to select the hardware-multiply-divide library (instead of SMPYD.LB) for later programs, or for your single merged library file. For now, if you want, you can create a CLI macro to do the load; or you can simply type in the filenames.

As RLDR processes MORTGAGE, you see the following series of messages on the console:

```
MORTGAGE.SV LOADED BY RLDR
REV xx.xx AT time date

.MAIN
I
FREAD
    .
    .
    .
NMAX  012177
ZMAX  000210
CSZE  000000
EST   000000
SST   000000
```

Following the initial line, which tells you the name of the save file and when it was created, RLDR describes all the modules that it extracted from the FORTRAN libraries and the system library to build the program. .MAIN is always the title of the main FORTRAN program; you can ignore the other modules. The next group of entries describes the memory addresses at which MORTGAGE will execute. NMAX is the highest address of normal-relocatable code plus one.

CSZE is the unlabeled common area size; here, there is none. EST and SST are the end and start of the symbol table, and, again, there is none. In most FORTRAN applications, you won't care about the information produced by RLDR; but if you plan to run dual programs or have limited memory, the NMAX and ZMAX figures may concern you. You can find more detail on this in your system reference manual or the *Extended Relocatable Loaders User's Manual*.

Executing the FORTRAN Program

You now proceed to the next step, executing MORTGAGE.SV. To execute this or any other program, simply type the program name and follow it with a).

```
MORTGAGE)
ENTER AMOUNT, RATE, YEARS
AND, 0 FOR SUMMARY OR 1 FOR
FULL SCHEDULE. SEPARATE EACH
ENTRY WITH A COMMA.
TERMINATE INPUT WITH RETURN.
```

You then respond with a request for the summary information (monthly payment only) given a mortgage of $20,000 at 9% for 25 years:

```
20000,.09,25,0)
```

MORTGAGE then types:

```
AMOUNT = $20000.00
INTEREST RATE = 0.0900
LOAN LIFE IS 25 YEARS

MONTHLY PAYMENT = $167.84
```

on the console, and then says:

```
TYPE 1 TO REPEAT, 0 TO STOP
```

To continue, type:

```
1)
```

Since you responded with 1, the same instructions appear on the console. This time, enter the same arguments but ask for a full schedule:

```
20000,.09,25,1)

AMOUNT = $20000.00
INTEREST RATE = 0.0900
LOAN LIFE IS 25 YEARS

MONTHLY PAYMENT = $167.84
NUM. INTEREST PRIN. PAY...
FATAL RUNTIME ERROR 15 AT LOC. xxxxxx
CALLED FROM LOC. yyyyyy
R
```

and find the CLI running on the console. The FORTRAN IV manual describes *Runtime Error 15* as a "field" error, which might have occurred in an F or E entry in a FORMAT statement. Checking the Fs in our program, we find a typo in statement 140: *F0.2* should be *F9.2*. Fix this with the Superedit command

```
J$CF0.2$F9.2$T$$
```

Recompile the program, and run it through RLDR again. Then execute MORTGAGE, and enter the figures:

20000,.09,25,1)

and the long listing begins. If the console is a display terminal, you can use CTRL-S, CTRL-Q to stop and start display. If you wrote the program for the line printer, the listing starts on the printer. The beginning of this summary appears in Figure 6-4; the interest total at the end of the whole summary is pretty appalling.

To terminate MORTGAGE and return to the CLI, type 0 to the REPEAT query. CTRL-A would do this at any time, but it wouldn't be as neat.

Compile, Load and Go

Instead of compiling a FORTRAN IV source program (FORT command), loading it (RLDR command), and executing it (programname command), you can use the CLG command to combine these steps. CLG is a program that directs the compiler, then the loader; then it executes the resultant save file. Before you can use CLG, you must merge the FORTRAN libraries into a single library named FORT.LB with the Library File Editor Merge command; this is described in an appendix of the *FORTRAN IV User's Manual*. Also, the source file, CLG.SV, and all other utilities involved must be in the current directory (or linked to it).

```
AMOUNT     = $ 20000.00
INTEREST RATE = 0.0900
LOAN LIFE IS  25 YEARS

MONTHLY PAYMENT = $    167.84

 NUM        INTEREST      PRIN. PAY        PRIN.BAL       INTEREST PAID TO DATE

  1       $   150.00      $    17.84     $ 19982.16        $   150.00
  2       $   149.87      $    17.98     $ 19964.18        $   299.87
  3       $   149.73      $    18.11     $ 19946.07        $   449.60
  4       $   149.60      $    18.25     $ 19927.83        $   599.19
  5       $   149.46      $    18.38     $ 19909.44        $   748.65
  6       $   149.32      $    18.52     $ 19890.93        $   897.97
  7       $   149.18      $    18.66     $ 19872.27        $  1047.15
  8       $   149.04      $    18.80     $ 19853.46        $  1196.20
  9       $   148.90      $    18.94     $ 19834.52        $  1345.10
 10       $   148.76      $    19.08     $ 19815.44        $  1493.86
 11       $   148.62      $    19.23     $ 19796.21        $  1642.47
 12       $   148.47      $    19.37     $ 19776.84        $  1790.94
 13       $   148.33      $    19.51     $ 19757.33        $  1939.27
 14       $   148.18      $    19.66     $ 19737.67        $  2087.45
                                  .
                                  .
```

Figure 6-4. Sample of Extended Schedule from MORTGAGE

End of Chapter

Chapter 7
Extended BASIC Programming

This chapter leads you through a sample session in Extended BASIC. It assumes that you have some experience with the BASIC language, and that you have loaded the BASIC System Generation program (BSG) from tape or diskette, and generated a BASIC system. Chapter 2, the *Extended BASIC System Manager's Guide* covers BSG. This chapter also assumes that you are the only person on the system using BASIC. On a multiuser BASIC system, log on according to your system manager. Skip step 1 below and all the CLI material in this chapter.

Aside from the *Extended BASIC System Manager's Guide*, we offer two other books on Extended BASIC: *basic BASIC*, for beginners, and the *Extended BASIC User's Manual*, which covers the features and commands of our BASIC. Here are the steps you follow to create a BASIC program:

1) Invoke the BASIC interpreter and get into BASIC by typing the CLI command:

   BASIC)

2) Write a series of BASIC program statements. BASIC has its own editor and an interactive compiler that rejects bad syntax as you type each statement.

3) Run the program with the BASIC command:

   RUN)

4) If the program runs correctly, you're done! Save the program on disk with the LIST command. Type BYE) to get back to the CLI.

5) If your program contains runtime errors, fix it using erroneous output or BASIC runtime error messages. Go to 3.

Writing BASIC Programs

You write a BASIC program as a series of statements, which you must begin with a number between 1 and 9999. Each statement includes a command to BASIC, which then executes the statements sequentially, by number; thus, your program can do useful work.

At various points, you can examine the statements in your program with the LIST command, or tell BASIC to execute the statements with the RUN command. BASIC's error messages will help you correct errors; you can correct offending statements by typing their line numbers, then the new text. When you're satisfied with a program, save it on disk with the command LIST "filename"); later, you can read it back into memory with the command ENTER "filename"). To print it on the line printer, type LIST "$LPT"). To start work on another program, type NEW), then proceed. To sign off BASIC and return to the CLI, type BYE).

You can execute a BASIC program only from BASIC; you can't do it from the CLI. The BASIC interpreter accepts both upper- and lowercase characters, and translates lowercase letters to uppercase. You can make lowercase text part of your program (in PRINT and comment statements) by editing your BASIC program file with Superedit, after the program runs.

Sample Session

This is a simple session with Extended BASIC. Bring up RDOS or DOS (if it isn't already running), sit down at the console, and try it. First, DIR to the partition or diskette that contains the BASIC directory (BASIC.DR). BASIC requires this directory to run, although you cannot invoke BASIC from inside it.

```
DIR proper-directory)
R
```

The person who generated the BASIC system could have named it anything he or she wanted. We're assuming that its name is BASIC, filename BASIC.SV.

```
BASIC)
EXTENDED BASIC REVISION xx.xx date
OPER .....
```

(Depending on your BASIC, it may request a password or device specifier information here. For password details, see the *BASIC User's Manual.* If it asks DIRECTORY SPECIFIER?, answer

```
BASIC)
```

Now, all BASIC programs you LIST to disk will go into directory BASIC.DR.

If BASIC.DR doesn't exist, you'll find yourself back in the CLI. Proceed to create it (CDIR BASIC), and type BASIC) again.

When BASIC comes up, it issues an asterisk (*) prompt. When this prompt appears, you can issue BASIC commands and statements.

```
*PRINT HELLO)
ERROR 02-SYNTAX
PRINT "HELLO")
HELLO
*
```

You must include string literals in quotes. For an explanation of other BASIC error messages, see Appendix A of the *Extended BASIC User's Manual.*

```
*10 REM THIS IS A REMARKS LINE.)
*20 READ A,B,C,D)
*30 PRINT D - A + B * C ↑ D)
*40 DATA 2,3,4,5)
*LIST)
0010 REM THIS IS A REMARKS LINE.
0020 READ A,B,C,D
0030 PRINT D-A+B*C ↑ D
0040 DATA 2,3,4,5
*RUN)
3075
END AT 0040
*
```

BASIC evaluated the expression as $[5-2 + (C^D * B)] = 3 + (4^5 * 3) = 3075$. The exponential character is an uparrow (↑) produced by SHIFT-6 or SHIFT-N.

Now, to store the little program on disk:

```
*LIST "ARITH1")
*
```

You can use any RDOS/DOS filename, with an extension if you want, for a BASIC program. Later, you can read this program back into memory by this filename:

```
*ENTER "ARITH1")
```

When you want to sign off BASIC, type:

```
*BYE)
(log off and accounting information)
R
```

Now get back into BASIC for the sample session:

```
BASIC)
EXTENDED...
*
```

Strings and Arrays

BASIC strings and arrays allow you to store and access alphanumeric strings and numbers by subscript. You can declare a string or array, assign it a fixed number of elements with a DIM statement, and then access elements by the string or array name, followed by the element subscript number in parentheses or brackets. String names begin with a letter and end with a dollar sign (e.g., AS, A1S); array names don't include the dollar sign (e.g., A,A1). For example, type:

```
*NEW)
*10 DIM N$(30))
*20 INPUT "WHAT'S YOUR NAME?",N$)
```

Run it:

```
*RUN)
WHAT'S YOUR NAME?
```

Type a name:

```
WHAT'S YOUR NAME? MELINA)
*
```

Line 20 uses a *string literal*, which you simply enclose in quotes, and BASIC accepts it literally; line 10 dimensions *string variable*, N$. Your response assigned values MELINA to the first six elements of N$, which look like this in memory:

```
 M    E    L    I    N    A    null .... null
N$(1) N$(2) N$(3) N$(4) N$(5) N$(6) N$(7) N$(30)
```

Elements $N(7) through $N(30) contain nulls (ASCII 000).

You can reference substrings using the form *stringname(element1, elementn)*. For example, append a statement to your program and run it:

```
*30 PRINT N$(1,3))
*RUN)
WHAT'S YOUR NAME? MELINA)
MEL
*
```

The subscripts specified elements 1 through 3; hence, MEL.

Now, we can try a more sophisticated version of the program, which dimensions two strings, compares them, and acts on the comparison. Note that whenever you change mode in a PRINT statement, from string literal or string variable or numeric variable, you must insert a semicolon (or comma). This rule accounts for the complex punctuation in lines 30 and 80. Type NEW), then type in this program. If you make a mistake, retype the bad line from the beginning.

```
10 DIM N$(30) O$(30))
20 INPUT "WHAT'S YOUR NAME?□",N$)
30 PRINT "IS IT REALLY□";N$;"?□TYPE IT AGAIN.")
40 INPUT O$)
50 IF O$<>N$ THEN GOTO 80)
60 PRINT N$;"□MUST REALLY BE YOUR NAME.")
70 STOP)
80 PRINT"IS IT□";N$;"□OR□";O$;"?"
```

Now, try running the program. If you want to save it on disk, type LIST "name"), where name is any legal RDOS or DOS filename. Chapter 8 of *basic BASIC* describes strings further.

BASIC arrays resemble strings, but each element in an array must be a number, whereas each element in a string must be an alphanumeric character. For example,

```
10 DIM A(20))
20 A(1) = 9.5)
30 A(2) = 46)
40 A(20) = A(1)+A(2))
50 PRINT A(1),A(2),A(20))
```

These statements dimension array A and assign values to three of its elements; the other elements contain nulls. Chapter 7 of *basic BASIC* explains elementary arrays.

BASIC Program

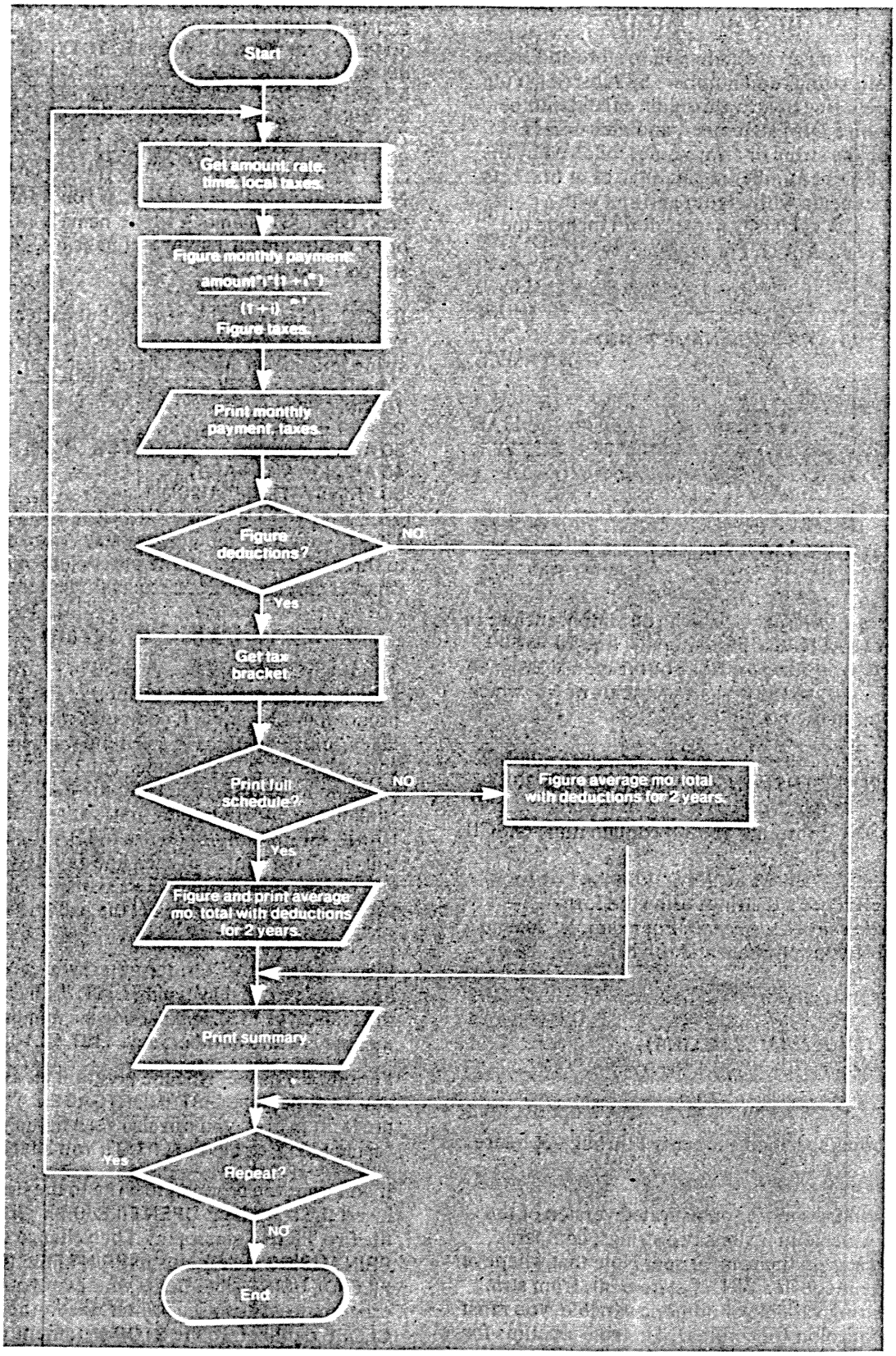
The BASIC program flowcharted in Figure 7-1 and shown in Figure 7-2 is a variation of the FORTRAN program in Chapter 6. It computes mortgage payments, taxes, and deductions in a general way; and writes its computations to the console. You can enter the program using Superedit from the BASIC directory (link to the Superedit files first), or you can use the BASIC interpreter. If you use the BASIC interpreter, it will check the syntax of each line as you type it in. If you use Superedit and make a syntax mistake, the BASIC interpreter will reject the bad line when you ENTER the program.

If you use the BASIC interpreter, you can examine the lines you've typed by typing LIST. To list a portion of the lines, type LIST number comma number, where each number is a line number, e.g., LIST 10,100).

Periodically as you type the program in, and when you're done, type LIST "MORTGAGE.BA", to write the program to disk; you can also get a hardcopy listing by typing LIST), or LIST "$LPT") if you have a line printer.

If you want the program to write to the line printer, insert the statement OPEN FILE (0,2), "$LPT" before the first PRINT statement. Then change the PRINT or PRINT USING statements to PRINT FILE (0), or PRINT FILE (0), USING statements wherever you want the program to write to the printer. Insert the statement CLOSE right before the STOP statement (line 300).

You should examine Figures 7-1 and 7-2 before proceeding to the next section.



SD-00919

Figure 7-1. MORTGAGE Program Flowchart

```
0010 REM    PROGRAM MORTGAGE.BA, COMPUTES MORTGAGE PAYMENTS, HAS TAX SUBROUTINE.
0020 PRINT "<12> I CALCULATE MORTGAGE PAYMENTS, INTEREST, AND TAXES."
0030 PRINT "TYPE AMOUNT OF PRINCIPAL, INTEREST RATE IN WHOLE NUMBERS,"
0040 PRINT "MORTGAGE LIFE IN YEARS, AND ANNUAL PROPERTY TAX BILL FOR HOUSE."
0050 PRINT "SEPARATE ENTRIES WITH A COMMA; FOR EXAMPLE  40000,10.5,25,2000."
0060 PRINT
0070 PRINT "  AMOUNT? RATE? YEARS? TAXES?"
0080 INPUT " ?   ",A,R1,Y,T
0090 REM   GET MONTHLY RATE R (R1/12), MAKE INTO FRACTION AS R1 WAS WHOLE NUMBER.
0100 LET R=R1/1200
0110 REM   GET NUMBER OF MONTHS M FOR LOAN.
0120 LET M=12*Y
0130 REM   COMPUTE MONTHLY PAYMENT.
0140 LET P=A*R*(1+R)↑M/((1+R)↑M-1)
0150 REM    DEFINE FORMAT F$ THAT ROUNDS NUMBERS TO NEAREST WHOLE CENT.
0160 LET F$="-------.##"
0180 REM    PRINT TOTALS AND GIVE OPTION FOR TAX SUBROUTINE.
0190 PRINT "MONTHLY PAYMENT:       TAXES:           HIDEOUS TOTAL:"
0200 PRINT USING F$,P,"  ",T," ",P+T
0210 PRINT
0220 PRINT "WANT TO COMPUTE THE TRUE COST AFTER U.S.TAX DEDUCTIONS ON THE"
0230 PRINT "INTEREST AND TAXES?  YOU MUST ITEMIZE TO QUALIFY."
0240 INPUT "ANSWER Y (YES) OR N (NO).       ",Q$
0250 REM  $ SPECIFIES STRING INPUT (E.G."Y") INSTEAD OF NUMERIC.
0260 IF Q$="Y" THEN GOSUB 1000
0270 PRINT
0280 INPUT "TYPE Y (YES) TO RUN PROGRAM AGAIN, ANYTHING ELSE TO STOP. ",Q$
0290 IF Q$="Y" THEN GOTO 0060
0300 STOP
1000 REM   TAX DEDUCTION COMPUTATION SUBROUTINE.
1010 INPUT "WHAT IS YOUR TAX BRACKET, IN WHOLE NUMBERS?    ",B1
1020 LET B=B1/100
1030 PRINT
1040 PRINT "SHOULD I LIST PAYMENTS FOR THE FIRST TWO YEARS? I HAVE"
1050 INPUT "TO FIGURE THE INTEREST ANYWAY. ANSWER Y (YES) OR N (NO).  ",Q$
1060 REM SET UP VARIABLES A1 (PRINCIPAL PD PER MONTH) AND I1 (FOR TOTAL INTEREST)."
1070 LET A1=A
1080 LET I1=0
1090 IF Q$<>"Y" THEN GOTO 1110
1100 PRINT "      MONTH    PRIN.     INT.    INT. TOTAL"
1110 REM   FOR-NEXT LOOP COMPUTES (OPTIONALLY LISTS) FIGURES BY MONTH AND TOTALS.
1120 FOR J=1 TO 24
1130   LET P1=A1*R/((R+1)↑M-1)
1140   LET I1=I1+(P-P1)
1150   LET A1=A1-P1
1160   IF Q$<>"Y" THEN GOTO 1180
1170   PRINT USING F$,J,P1,P-P1,I1
1180 NEXT J
1190 PRINT
1200 REM GET DEDUCTIONS D FOR 1 YEAR, T(AXES) + I1/2 (HALF OF 2 YRS. INTEREST)
1210 LET D=T+I1/2
1220 PRINT "ANNUAL MORTGAGE-RELATED DEDUCTIONS ARE:"
1230 PRINT USING F$,D
1240 PRINT "BUT I MUST SUBTRACT THE $3200 STANDARD (0 BRACKET) DEDUCTION"
1250 PRINT "BUILT INTO THE TAX TABLES. THE TRUE MONTHLY COST IS:"
1260 LET D1=D-3200
1270 REM GET REAL MO. COST. (TOTAL MO. PAY =  P+T12) - ((BRKT * ADJ. DEDS)/12)
1280 LET C=(P+T/12)-(B*D1/12)
1290 PRINT USING F$,C
1300 PRINT "                    ****SUMMARY****"
1310 PRINT "     LIFE:   AMOUNT:      RATE:  CASH PAY:   BRKT:   TRUE COST:"
1320 PRINT USING F$,Y,A,R1,P+T/12,B1,C
1330 RETURN
```

Figure 7-2. MORTGAGE Program With Errors

Running the BASIC Program

To run the program, get into BASIC (if you're not already there), and type:

```
*NEW)
*
ENTER "MORTGAGE.BA")  (or whatever name and
                       extension you gave it)

*RUN)

I CALCULATE MORTGAGE PAYMENTS,
INTEREST, AND TAXES.
TYPE AMOUNT OF PRINCIPAL, INTEREST RATE
IN WHOLE NUMBERS.
MORTAGE LIFE IN YEARS, AND ANNUAL
PROPERTY TAX BILL FOR HOUSE.
SEPARATE ENTRIES WITH A COMMA; FOR
EXAMPLE 40000, 10.5, 25, 2000.
AMOUNT? RATE? YEARS? TAXES?
?
```

Respond with some plausible figures:

```
40000, 10.5, 25, 2000)
```

and the program responds:

```
MONTHLY PAYMENT:   TAXES:  HIDEOUS TOTAL:
      377.67       2000.00       2377.67
WANT TO COMPUTE THE TRUE COST...
...ITEMIZE TO QUALIFY.
ANSWER Y (YES) OR N (NO).
```

This seems a little high -- over $2,000 per month. We forgot to convert the yearly tax figure to months. (Depending on the precision generated on your BASIC system, your answers may differ slightly.) It would be meaningless to proceed so type:

```
N)
TYPE Y (YES) TO RUN PROGRAM AGAIN,
ANYTHING ELSE TO STOP. )
STOP AT 300
*
```

The problem is in line 200, and you can fix it by dividing the tax variable, T, by 12. Replace the existing line 200 by typing:

```
200 PRINT USING FS,P,"□□",T/12,"□",P+T/12)
```

and run it again:

```
RUN)
I CALCULATE... I

AMOUNT? RATE? YEARS? TAXES?
? 40000,10.5,25,2000)
MONTHLY PAYMENT:   TAXES:  HIDEOUS TOTAL:
      377.67       166.67       544.34
WANT TO COMPUTE.....
ANSWER Y (YES) OR N (NO).
```

These figures are more reasonable so you can proceed with the program:

```
Y)

WHAT IS YOUR TAX BRACKET, IN WHOLE
NUMBERS?
```

The tax bracket issue is explained later in this chapter. For now, try 25, a typical bracket:

```
25)

SHOULD I LIST PAYMENTS FOR...
...ANSWER Y (YES) OR N (NO). Y)

MONTH     PRIN.     INT.      INT. TOTAL
1.00      27.67     350.00    350.00
2.00      27.65     350.02    700.02
3.00      27.64     350.04    1050.06
4.00      27.62     350.06    1400.11

MORTGAGE-RELATED DEDUCTIONS ARE:
6202.63
BUT I MUST SUBTRACT THE $3200...
...TRUE MONTHLY COST IS:
481.79

                ****SUMMARY****
LIFE:AMOUNT:RATE:CASH      PAY:BRKT:TRUE
COST:
25.00 40000.00  10.50  544.34      25.00  481.79
TYPE Y (YES) TO RUN PROGRAM AGAIN.....
```

It works! At least it appears to work. (Clearly, if the TRUE PAY figure exceeds the CASH PAY figure, it may not pay you to itemize.) Let's examine the interest schedule again, though--because the program bases its deductible figure on the last value of variable I1 in this schedule.

Unfortunately, there is a problem in this schedule--the amount of interest paid each month increases while it should decrease. This must be wrong, and being wrong, it voids the entire deduction figure. Looking over the FOR-NEXT loop that computes the monthly interest, note that we forgot to decrement the month indicator, N, for each circuit through the FOR-NEXT loop. You can fix this easily. First, stop the program:

```
)
STOP AT 300
•
```

Having incremented line numbers by 10, you can easily insert a new statement:

```
1155 LET M=M-1 )
```

Now RUN) it again, giving the same figures (40000, 10.5, 25, 20, and 2000) and tax bracket (25), to compare the results. The monthly schedule now says:

MONTH	PRIN.	INT.	INT. TOTAL
1.00	27.67	350.00	350.00
2.00	27.92	349.76	699.76
3.00	28.16	349.51	1049.27
4.00	28.41	349.27	1398.54

```
MORTGAGE-RELATED DEDUCTIONS ARE:
6164.34
BUT I MUST SUBTRACT...
...TRUE MONTHLY COST IS:
482.58
```

```
                ****SUMMARY****
LIFE:AMOUNT:RATE:CASH        PAY:BRKT:TRUE
COST:
25.00 40000.00  10.50  544.34      25.00  482.58
TYPE Y (YES) TO RUN PROGRAM AGAIN.....
```

Eureka! The interest is now declining, and the program works correctly. (The difference may not seem significant here, but if the FOR-NEXT loop covered 60 months instead of 24; i.e., J = 1 TO 60; it would be immense.)

You've fixed the program, so you can write it to disk under its original name; this overwrites the old, erroneous version:

```
LIST "MORTGAGE.BA")
•
```

You can also print it (LIST "SLPT") if you have a line printer, or type it (LIST). To leave BASIC and return to the CLI, type:

```
BYE)
    .
    .
R
```

All programs you write via the BASIC interpreter reside in the BASIC directory; the interpreter automatically goes to this directory when you invoke it.

Itemized Deductions and Tax Bracket

For simplicity, this 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 ($3200 if married filing jointly, $2200 if single). The standard deduction is already figured into the tax tables. This is why line 420 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 "3200" in line 1260. 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 1260; e.g., LET D1 = D+Q-3200.

In any case, if you are single, change the 3200 in line 1260 to 2200; e.g., LET D1 = D-2200.

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 3200 (if married filing jointly) or 2200 (if single); this is your gross income less deductions.

Multiply the number of exemptions you used to find your tax by 750, and subtract the product from the original figure. This is roughly your net income.

Now look at the Tax Rate Schedules (not the Tax Tables) in any 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 a tax-bracket 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 3,200 is 19,800. 19,800 minus 3 x 750 is 17,550. In Tax Rate Schedule Y, you find that 17,550 falls between $15,200 and $19,200. Reading to the right on the same line, you find $3,260 + 25%. Thus your tax bracket is 25% because the government took a quarter from the last taxable dollar. Thus you save a quarter for each dollar the new mortgage lets you deduct. (This procedure yields only an approximation -- because the mortgage deductions might drop you into a lower bracket, thus save you less in deductions -- but it is a fairly close approximation.)

End of Chapter

Chapter 8
Assembly - Language Programming: The Assemblers

This primer can't possibly teach you all you must know to program in 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 instruction set for your computer. But even if you do not, you will profit by reading this chapter, for it will familiarize you with 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.

Before it can do useful work, a program must be translated into machine-level instructions. System software does this: programs like the FORTRAN compiler, the Relocatable Loader, and the BASIC interpreter, translate your program commands into machine code: binary numbers that physically direct the computer. Assembly language is no exception; it also employs symbolic commands which translate into machine code. The software that does the translation is called an assembler.

The Assemblers

You can choose between two assemblers: the Macroassembler (MAC) and the Extended Assembler (ASM). MAC is more powerful and more flexible, while ASM is faster. Each assembler has a manual of its own, which you can read for more detail. For the concepts we'll study in this chapter, and the program we'll write in the next, there is no practical difference between the two. We have chosen to emphasize the Macroassembler (MAC); but if you want to use ASM, go ahead. The term *assembler* in this chapter and the next applies to either MAC or ASM.

To produce a program, you start by coding a source file in assembly language and emerge with a save file. The sequence looks like this:

```
SOURCEPROGRAM
     ↓
   MAC or ASM
     ↓
SOURCEPROGRAM.RB
     ↓
   RLDR
     ↓
SOURCEPROGRAM.SV
```

A source program (also called a module) employs symbolic instruction codes (such as LDA 0,2 for "load the contents of location 2 into accumulator 0") and operating system calls (such as .RDL for "read a line"). Your modules will also use assembler pseudo-instructions (pseudo-ops), which direct the assembly process but do not result in any final program instructions themselves.

Each 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 module

The assembly listing shows your original source module(s) and additional information such as the octal codes of your instructions and data, the absolute or relative locations of these items in the executable program, and other miscellaneous information.

Assembly errors are analagous to FORTRAN compile-time errors. They usually indicate syntactical errors, not errors of logic in your program. Unless you send them to another file, error messages always go to the console. The listing will also contain error codes beside each offending source line.

After you have produced a binary module with no assembly errors, you then process it (perhaps along with other modules) with RLDR (the Relocatable Loader) to produce an executable program. You then run the program, and if it does not work the way you want it to, you examine it with the debugger.

The debugger lets you run any part of a program, stop it temporarily so that you can examine the contents of memory locations or accumulators, and make temporary minor changes to the program. You will probably have to repeat the process of assembly, processing with RLDR and execution/debugging several times before you produce a program that works as you want it to.

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 assembler accepts free-form input, hence you need not place any of these elements in any specific line position. The only requirement for source code is that if they appear in any line, the order of these elements must be label, instruction, arguments, and comments. Each label ends with a colon. For our purposes, the assembler considers only the first five characters of each label to be significant (it cannot, for example, differentiate between the labels BEGIN and BEGINNING).

Each comment begins with a semicolon and extends to the end of the line. The assemblers do not process comments; they merely pass them on intact to the program listing. Figure 8-1 shows you a page from an assembly listing.

Notice that there are three columns of numbers; to the right of these, you can see the source code that was input to the assembler. 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/ comments
          argument

AGAIN:  .SYSTM      ;THIS IS
        .GCHAR      ;GET
        JMP ER
```

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; the assembler requires only that you follow the sequence of label (if any), instruction, arguments (if any), and comments (if any) within a line.

Figure 8-1 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. The assemblers use page and line numbers in the alphabetical cross-reference they produce with each program listing (shown in Figure 8-2). Now return to Figure 8-1. 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 017) or they may indicate only a relative position (the 3rd location from the beginning of this series). Columns 10 through 15 show the (unrelocated) octal assembled value that will reside in this location, if this value is know when the program is assembled. If the assembler doesn't know this value, perhaps because it references another module, the listing shows the contents as 000000.

The two remaining columns, 9 and 16, contain relocation flags. In essence, relocation applied to an address (column 9) indicates that RLDR, not the assembler, will assign a final address for the location, since only it can determine where to place the code. Relocation applied to data indicates where data or arguments are found. The symbols 'and" indicate the types of relocatability. Relocation is described in greater detail in the appropriate Assembler user's manual.

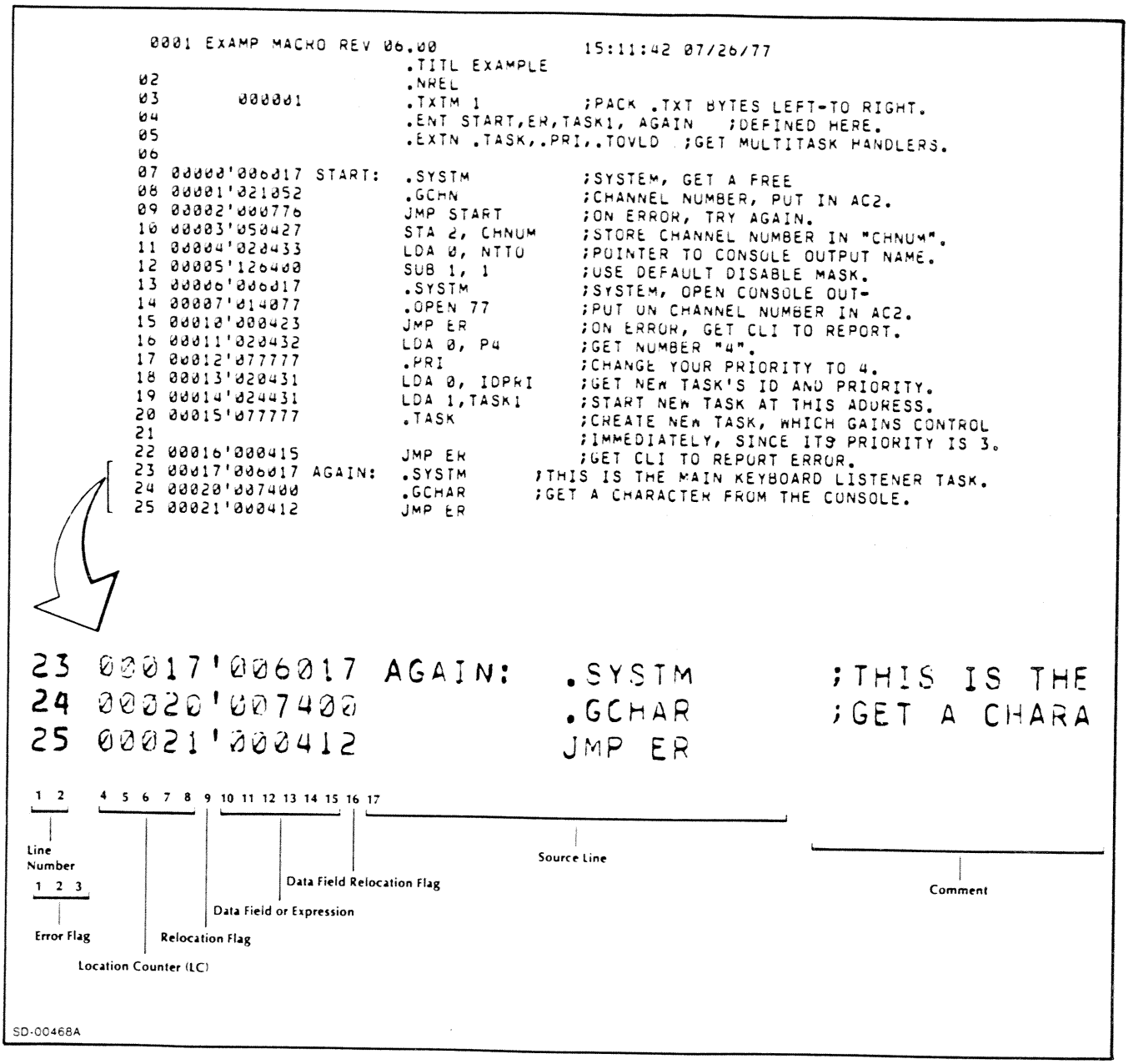


Figure 8-1. Program Listing

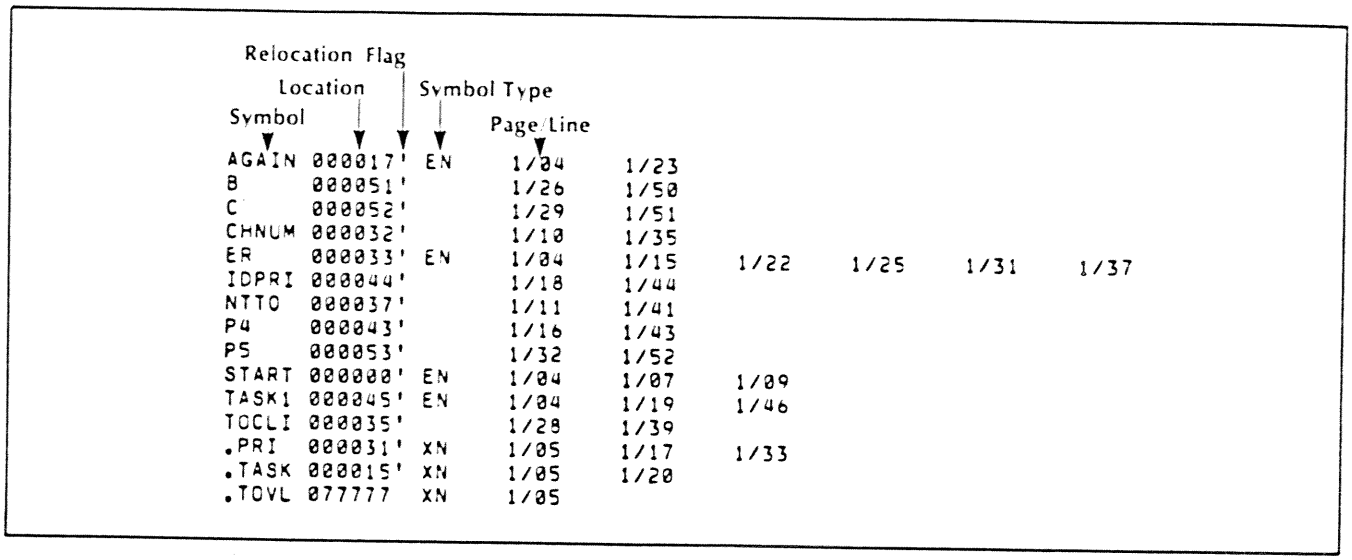


Figure 8-2. Cross-Reference Listing

Figure 8-2 shows the cross-reference from the entire program whose listing begins in Figure 8-1.

The cross-reference lists the page and line number where each user symbol (such as a label) is defined, and the page and line number of each reference to this symbol. The column labeled "symbol type" indicates which symbols were externally defined or entered. We will explain symbol types when we discuss pseudo-ops.

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 in their possible arguments (see *Special Instruction Symbols* described below). Other symbols are assembler pseudo-ops; also described below. Finally, you will want to create your own symbols. For example, you will want to name memory locations (e.g., START for the beginning of your program) so that 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 character is usually a space, tab, or carriage return.

The Symbol column of Figure 8-2 shows some symbols. Others are:

```
.DATA
DATA
FILE1
BT000
MESS1
.MSG1
```

While your symbols may be longer than five characters, for our purposes the assembler considers only the first five to be unique. Generally, your symbols should be five characters or less.

You may not use a period (.) as a single-character symbol. The assembler 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.

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., ERDNE) defined in certain system parameter files. You can always clear up such errors by simply modifying the offending symbol, perhaps by placing a period at its beginning or end. Assembly errors are described in an appendix of the appropriate Assembler user's manual.

Argument Operators

From time to time, you may want to perform some operations on symbols. Typical operations include forming a byte pointer from a symbol name, and creating indexes in a table. For example, if START is the name of the beginning of a text string, 2 times START would be a byte pointer to the beginning of this string (the next chapter reviews byte pointer and byte string concepts). Similarly, if TABLE is the name of the beginning of a table with one-word entries, then TABLE+1 would name the second entry in the table, TABLE+2 the third entry in the table, etc.

You may apply the following operators to symbols, integers, and the current location counter:

	Operator	Meaning
Arithmetic	+	Addition (2+3) or unary plus (+3)
	-	Subtraction, (5-2) or unary minus (-7)
	*	Multiplication
	/	Division

Thus you could write JMP .+5 to jump 5 words from the current location. If your program reserves a series of addresses whose first word is address TABLE, indexes off TABLE wold be TABLE+1, TABLE+2, etc.

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.

Instruction Types:

Most Data General 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. The address they reference must fit in 8 bits, because their address field is 8 bits long. Examples of MRIs are LDA 0, TEMP (load accumulator 0 with the contents of address identified with label TEMP) JMP ER (Jump to the instruction identified by label ER), and JSR PRINT (Jump to subroutine PRINT, save return address in AC3). Some basic MRI instructions and their *simplest* formats are shown in Table 8-1.

- *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 8-2.

- *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 8-1. Common MRI Instructions

Instruction and Format	Explanation	Example
JMP addr	Jump to address addr, which can be a symbolic or numeric address or expression. This definition of addr applies to all instructions.	JMP ER JMP LOOP+6 JMP 300
JSR addr	Jump to addr, save return address (current addr +1) in AC3. You can return from a JSR with a JMP 03 instruction (if the code to write you jumped didn't overwrite AC3).	JSR SUBR1
LDA ac addr	Load accumulator (ac) with contents of addr.	LDA 0, MYFIL
STA ac addr	Store the contents of ac in addr.	STA 3 LOOPO
DSZ addr	Decrement the contents of addr by 1, skip the next instruction if contents equal 0.	DSZ COUNT

Table 8-2. 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 or more of the carry, shift, or no-load options; *s* is an optional skip directive.	ADD 0 2
SUB *[opt]* acs acd *[s]*	Subtract the contents of acs from acd, place result in acd.	SUB 0 , 0
MOV *[opt]* acs acd *[s]*	Copy the contents of acs into acd.	MOV 2, 0
AND *[opt]* acd acs *[s]*	Logically AND the contents of acs with acd, place the result in acd.	AND 0 1

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 8-2.

You can append *one* of the following one-character symbols (opt) as a suffix to arithmetic/logical (ALC) instructions to produce the following results:

Opt	Operation	Result
C	complement carry	Set the carry bit to the opposite of its current state before the arithmetic/logical operation.
L	left shift	Combine the carry bit and the operand, and shift the 17-bit result one bit left, after the arithmetic/logical operation.
O	set carry	Set the value of the carry bit to 1 before the arithmetic/logical operation.
R	right shift	Combine the carry bit and the operand, and shift the 17-bit result one bit right, after the arithmetic/logical operation.
S	swap	Swap the left and right bytes for the arithmetic/logical operation.
Z	clear carry	Set the value of the carry bit to 0 before the arithmetic/logical operation.

You may use the following 3-character skip symbols with arithmetic/logical instructions to perform single-word skip operations. These are shown as *s* in example formats. You must always place these symbols at the end of the 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 9.

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 memory reference instruction (such as LDA, load an accumulator; or JSR, jump to a subroutine, save return address), or before an expression, you may use the commercial at sign (@). An @ anywhere in a memory reference instruction 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 accumulator 1 (AC1) with the contents of memory location 60; the second loads AC1 with the word whose *address* is in location 60. Likewise, using @ in a data word sets bit 0 (the indirect bit for a data word) of that word to one:

```
000025 25
100025 @25
```

Like @, a number sign (#) may appear anywhere in an arithmetic/logical instruction. A # sets bit 12 of the instruction to 1; this is the no-load bit, and you use it with one of the arithmetic/logical instruction symbols 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, places the result in 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.

The following examples show you 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.

Note that whenever you use the no-load (#) symbol you must specify one of the skip symbols (SZR, etc.) and specify a shift operation (L for shift left, R for shift right, etc.). The shift operation can be a dummy (e.g., SUB C,0,0,SZR when you don't care about the carry bit) but it must be present for no-load to work.

Pseudo-ops

A *pseudo-op* instruction directs the operation of the assembler. It is called a "pseudo instruction" because your program never executes it. The pseudo-ops that you need to start assembly language programming are:

Pseudo-op	Function
.BLK	Reserve a series of 16-bit words.
.END symbol	Terminate a module and name a starting address
.ENT	Declare an entry point or symbol to be available for other modules' use.
.EXTD	Declare a symbol (unsigned 8-bit or signed 7-bit quantity) or page-zero (ZREL) entry point to be external; that is, found in some other module. .ZREL references are useful because they will fit into the 8-bit field of an MRI instruction.
.EXTN	Declare an entry point or 16-bit symbol to be external; that is, found in some other module.
.NREL	Assemble the following code and data for execution in normal-relocatable (NREL) memory.
.TITL	Assign a title to a module.
.TXT	Create an ASCII text string.
.TXTM	Specify text string byte packing (e.g., left to right).
.ZREL	Assemble the following code and data for execution in page zero (ZREL).

Often you will want to refer to a symbol or call a subroutine which is defined outside the module you are writing. To do this, you must name the symbol or entry point in an .EXTD or .EXTN pseudo-op. .EXTD symbols are limited to 8 bits, and .EXTN symbols are 16 bits long. That is, you could write

```
.EXTD A
LDA 0,A
```

without receiving an assembly error, but

```
.EXTN A
LDA 0,A
```

would receive an assembly error (LDA is an MRI instruction with an 8-bit displacement field). Each argument that you name in an .EXTN or .EXTD pseudo-op must be made available with an .ENT pseudo-op in another module that you will load with this module.

The .TXT pseudo-op assembles a character string into its equivalent ASCII codes (two per word), terminated by a null byte packed right to left. Insert a ".TXTM 1" pseudo-op at the beginning of the program to pack bytes left to right. You can use any character not in the string as string delimiters. Thus,

```
.TXTM 1
      .
      .
      .
.TXT "ABCDE"
```

stores the ASCII codes for A and B in the first word, C and D in the second, and E < null > in the final word. Text (.TXT) strings can contain lowercase letters.

.TITL assigns a title to a module (do not confuse the title with the module's filename). The module's 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 storage extends through 377_8, and is directly addressable by all instructions. .NREL starts assembly storage at the beginning of NREL memory.

Now that we have described certain assembler pseudo-ops, we can explain the third column of each assembly-listing cross-reference (see Figure 8-2). This column contains two-letter codes describing the types of symbols which have been the argument of certain pseudo-ops.

Code	Meaning
EN	entry (.ENT pseudo-op)
XD	external displacement (.EXTD pseudo-op, not shown in Figure 8-2)
XN	external normal (.EXTN pseudo-op)

There are other codes, outside the scope of this discussion, which 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.

Examples:

```
TABLE:.BLK 10.
```

This example reserves a block of ten memory words; the first word in the series has the symbolic name TABLE, the second TABLE + 1, and so on.

.END

Indicate the End of a Module

Format:

.END *[expression]*

Description:

This pseudo-op terminates each assembly language module. If the program you are building has several modules, one of these modules must supply an *expression* argument to .END indicating the address to receive control when the program is executed. ASM requires a .END for each module; MAC does not.

Examples:

```
START: SUB 0,0
       .
       .
       .
       .END START
```

In this example, the .END pseudo-op terminates the module and defines START as the address that will receive control when you run the program.

.ENT

Define a Module Entry

Format:

.ENT symbol *[symbol...]*

Description

This pseudo-op declares symbols defined within the module to be available for referencing by other separately-assembled modules. Each symbol must not only be unique from other symbols within the module (this is always a requirement), but it must be unique from entries defined in other modules that you process together to form a single program.

Other modules use the .EXTN pseudo-op to reference symbols defined by the .ENT pseudo-op.

In this example, PROGA is the main program. It can call two external routines in module TRIG: SINE and COS. The pointer .SINE is required because JSR is a memory reference instruction with an address field of 8 bits. .SINE must be within 127 words of the JSR. SINE can be anywhere in the program since the word SINE in .SINE:SINE has 15 bits available for addressing.

Examples:

```
        .TITL PROGA      ; PROGA is main module.
        .EXTN SINE,COS   ; SINE, COS are in module TRIG.
        .TXTM 1
        .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        ; Address in othe module.
.COS:   COS         ; Ditto.
        .END START


        .TITL TRIG      ; TRIG does trig.
        .ENT SINE, COS  ; Identify entries.
        .TXTM 1
        .NREL

SINE:   STA 3, USP  ; Save return addr in USP,
                    ; safe ZREL address
                    ; described later.
         .          ; SINE processing routine
         .          ; is here, uses AC3.
        LDA 3, USP  ; Done, restore return addr.
        JMP 0, 3    ; Return to PROG1.
COS:    STA 3, USP  ; Save return.
           .        ; Do COS processing.
        LDA 3, USP  ; Restore return addr.
        JMP 0, 3    ; Return to PROG1.
        .END
```

.EXTD

Declare an External Displacement

Format:

.EXTD symbol *[symbol...]*

Description:

This pseudo-op declares that one or more symbols referenced by this module are defined (.ENTered) in other modules. The value of the symbol must be an 8-bit quantity; e.g., a ZREL address. Generally, you will use .EXTD when the symbol is defined in another module's ZREL space.

Here, the ERR routine in module MAIN is available to module MOD1 through a pointer in MAIN's ZREL space. This allows the assembler to resolve ERR's address for the JSR instruction (which it could not otherwise do because JSR's address field is only eight bits). All of a program's ZREL space (locations $50\text{-}377_8$) is accessible to the eight bits of an MRI instruction.

Example:

```
                .TITL  MAIN ; Main program.
                .ENT  .ERR    ; Error handler is here.
                .EXTN FOO     ; FOO code is in other module.
                .TXTM 1

                .ZREL
.ERR:           ERR           ; ERR defined in MAIN's ZREL.

                .NREL
START:          LDA 0 ... ; MAIN's NREL
                 .        ; code is here.
                 .
ERR:             .        ; Error handler.
                 .
                .END START


                .TITL MOD1    ; Module MOD1.
                .ENT FOO
                .EXTD .ERR    ; .ERR defined in
                              ; another module's ZREL.
                .TXTM 1
                .NREL
FOO:            LDA ...       ; MOD1's code is
                 .            ; here.
                JSR @ .ERR    ; Use .EXTD error
                              ; handler.
                 .
                .END
```

.EXTN

Declare an External Reference

Format:

.EXTN symbol *[symbol...]*

Description:

The pseudo-op declares that one or more symbols referenced by this module are defined (.ENTered) in other modules. Note that the symbol can be a 16-bit quantity.

Examples:

See the .ENT example.

.NREL

Set the Location counter to Normal-relocatable Code

Format:

.NREL

Description:

This pseudo-op sets the location counter to the next available word of normal-relocatable memory. It is normally required in each module.

Examples:

```
        .TITL PROGB
        .TXTM 1
        .ENT INIT
        .EXTN SORT3
        .NREL ;Normal relocation from now on.
INIT:   LDA 0, FILEA
        .
        .
        .
        .END INIT
```

.TITL

Entitle a Binary Module

Format:

```
.TITL symbol
```

Description:

This pseudo-op gives an RB 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 relationship to the module's filename, although you might want to use the same names for clarity.

Examples:

```
.TITL EXAMPLE

.NREL
      .
      .
      .
.END
```

This example assigns the title EXAMPLE to this module and prints EXAMPLE at the top of each listing page. (The assembler, 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".)

.TXTM

Specify .TXT Byte Packing

.TXT

Create a Text String

Formats:

```
.TXTM number
.TXT u string u
```

Description:

Normally, the assembler packs bytes right to left - which is the wrong order for ASCII text strings and other required items. To correct this, insert a .TXTM 1 statement before the .TXT appears in your program.

The .TXT pseudo-op creates an ASCII string, which can contain any ASCII characters. 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 them within angle brackets.

Examples:

```
.TXTM 1
      .
      .
      .
.TXT "ABCDE"
```

This example creates the ASCII string consisting of three words. The first contains the letters AB, the second contains the letters CD, and 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 AB and CDE, separated by a horizontal tab (ASCII code 011).

.ZREL

Set the Location Counter to Page Zero

Format:

.ZREL

Description:

This pseudo-op sets the location counter to the next available word in lower page zero. This is initially location 50_8.

Example:

```
                        .TITL PROGC
          000001        .TXTM 1

                        .ZREL
00000-065554 .ER:       ERR             ; ZREL pointer to
                                        ; error  handler.

                        .NREL           ; NREL code begins.
00000 020454 INIT:      LDA 0, NW
00001 006017            .SYSTM          ; System call
00002 014000            .OPEN 0         ; always needs
00003 002000            JMP @ .ER       ; error return.
   .      .              .             ; Body of program.
   .      .              .             ;
65554 006017 ERR:       .             ; Error handler code
                        .             ; begins at addr 65554,
                                      ; yet is accessible via
                                      ; ZREL from anywhere in
                                      ; 32K; e.g., location 3.
   .      .             .
                        .END INIT
```

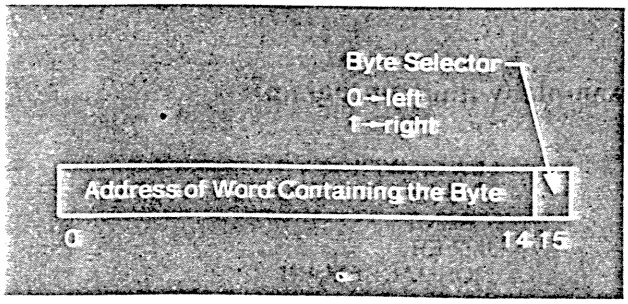
End of Chapter

Chapter 9
Programming RDOS or DOS Assembly Language System Calls

This chapter first introduces some operating system calls, with examples, and then shows the examples coded into a viable program. We'll then write an assembly language program, assemble it, correct assembly errors, process it with RLDR, and execute it. We will finish this chapter by showing you how to debug the program.

Operating System Conventions

Most Data General computer words are 16 bits long, and bit positions are numbered left to right, 0 to 15 inclusive. A byte is 8 bits long. A byte string consists of a sequence of bytes, packed right to left in series of one or more words. A byte pointer consists of a single word with two fields (see Figure 9-1). To pack bytes left to right, as required for system calls, insert a .TXTM 1 statement at the beginning of the program.



SD-00526

Figure 9-1. Byte Pointer Structure

The left field (bit positions 0 through 14) contains the address of the word that holds the selected byte. When right field (bit 15) equals 1, the pointer selects the right half, or byte; when this bit equals 0, the pointer selects the left byte. You can produce a byte pointer by multiplying the address at which it begins by 2; this shifts all bits left and zero (which specifies the left byte) in bit 15.

Here is a byte pointer example:

```
.TXTM 1
      .
      .
      .
BPTR:         .+1*2
              .TXT "$TTI"
```

.+1 is the address of the next location - the start of text string $TTI. The label BPTR contains the start address of the text string. The multiplier *2 zeros bit 15, thereby specifying the first byte, $.

System Call Format

Code each system call in the form:

```
.SYSTM
call-name
error return
normal return
```

You must precede the call with the mnemonic .SYSTM, and reserve a word after the call for the error return, which receives control on either an error condition, or an unusual condition such as an end-of-file. You must always reserve an error return word, even if you envision no possible exception condition, or even if no condition is currently defined. If no error or exception condition exists, control goes to the normal return.

Whenever control goes to the error return, the system places a numeric error code in AC2. These error codes are defined in the user parameter file. PARU.SR and they are explained in Appendix A of your system reference manual.

Upon either an error return or a normal return, AC3 contains the current contents of location 16_8, the user stack pointer. You can use this location as you want. Unless the text indicates otherwise, all other accumulators will remain unchanged.

Some system calls require you to pass a byte pointer (Figure 9-1) in AC0; some of these need other data in AC1.

All *system* calls are executed in system address space; *task* calls, not described here, are executed in your own space. You must not use memory locations $0\text{-}15_8$ or 17_8; the system uses these locations for call processing.

Operating System Call Summaries

As we have done throughout this book, we will describe only the most important aspects of these calls. After you have mastered the techniques for using these simple versions, you may want to use the more complex versions. For many applications, however, the simple versions of these calls will suffice.

The calls we describe here are:

.CRAND Create a random file.

.OPEN Open a file or device for reading or writing.

.APPEND Open a file or device for appending.

.RDL Read a line.

.WRL Write a line.

.RTN Stop the program and return to the CLI.

If your program will execute I/O transfers, it must follow this sequence of operations:

1) Open the file or device on a channel via an .OPEN or .APPEND call. The program will use the channel number, not filename or device name, for all I/O to the file or device.

2) Read or write data to the channel (.RDL or .WRL)

.RTN will close open files, terminate the program, and return to the CLI.

If a call takes the error return to your program, AC2 will contain an appropriate error code. The sample program we've devised uses system call .ERTN to ask the CLI to print an appropriate error message on the console.

.CRAND

Create a Random File

.CRAND creates an empty random file in the current directory.

Format:

```
.SYSTM
.CRAND
error return
normal return
```

Required input:

AC0 - byte pointer to the filename of the new file.

Return:

AC0 - unchanged.
AC1 - unchanged.
AC2 - error code.

Example (within a program):

```
        .
        LDA 0,MYFIL
        .SYSTM
        .CRAND
        JMP ER
        ;NORMAL RETURN.

MYFIL:  .+1*2           ;BYTEPOINTER.
        .TXT "OUTPUT"
ER:     .SYSTM

        .ERTN           ;.ERTN TELLS THE CLI TO
                        ;DISPLAY THE REASON
                        ;FOR THE ERROR RETURN.
```

.OPEN

Open a File for Reading or Writing

.APPEND

Open a File for Appending

.OPEN associates a file with an I/O channel and opens it for any kind of I/O. .APPEND associates a file with an I/O channel and opens it for writing only. If the file is an output device like a line printer, .OPEN and .APPEND are practically identical. For a disk file, if you open via .OPEN, lines are written to the beginning of the file. If you open via .APPEND, reads aren't allowed and lines are appended to the file.

While the file is open, your program references it by the channel number assigned in the open call. It keeps the channel number until the program issues .RTN or .CLOSEs it. We do not describe .CLOSE in this book because the .RTN or .ERTN calls automatically close all channels.

You can allocate a number of channels (and tasks) to a program in the RLDR command line. If you omit channel and task data entirely, the program receives one task and eight channels; this will suffice for the program in this chapter.

Format

```
.SYSTM
.OPEN n    ; n is the I/O channel number
  or
.APPEND n
error return
normal return
```

Required input:

AC0 - byte pointer to the file or device to be opened.

AC1 - if the file is a device, AC1 acts as a characteristic disable mask. You'll probably want to keep the default characteristics by setting AC1 to 0 (via a SUB # 1,1 instruction).

Return:

AC2 - error code.

Example (within a program):

This program just created the file "OUTPUT".

```
        LDA 0, NTTI     ;BYTE POINTER TO
                        CONSOLE
                        ;INPUT FILENAME.
        SUB 1,1         ;USE DEFAULT MASK FOR
                        OPEN.
        .SYSTM          ;0 IS THE I/O CHANNEL
                        NUMBER-
        .OPEN 0         ;WHILE THE $TTI IS
                        ;OPEN ON 0,
                        ;YOU'LL ADDRESS IT
                        ;AS 0.
        JMP ER          ;ER WAS DEFINED
                        EARLIER.


        LDA 0, OUTPUT
        SUB 1,1         ;DEFAULT MASK.
        .SYSTM
        .APPEND 1       ;OPEN "OUTPUT"
                        ;FOR APPENDING ON
                        ;CHANNEL 1.
        JMP ER


NTTI:   .+1*2
        .TXT "$TTI"     ;FILENAME $TTI.

OUTPUT: .+1*2
        .TXT "OUTPUT"
```

.RDL

Read a Line into Memory from a File or Device

Use this command to read an ASCII line from a file opened on a specified channel into a memory area (which should be 133 bytes long). Reading will terminate when the system reads a carriage return, form feed, or null. If the program is reading from the console keyboard (filename $TTI), you can indicate an end-of-read by typing).

Format:

```
.SYSTM
.RDL n    ; Read from the file on channel n.
error return
normal return
```

Required input:

AC0 - byte pointer to the 133-byte line buffer area.

Return:

AC1 - byte count of characters read, not including terminator.

AC2 - error code.

Example (within a program):

This program just .OPENed the TTI and opened "OUTPUT" for appending.

```
        .
        .
        LDA 0, SPACE
        .SYSTM
        .RDL 0
        JMP ER          ;DEFINED EARLIER.

SPACE:  .+1*2           ;RESERVE 67 WORDS
                        (134 BYTES)
        .BLK 67.        ;FOR THE LINE. THE
                        PERIOD
                        ;INDICATES A DECIMAL
                        FIGURE
                        ;TO THE ASSEMBLER.
```

.WRL

Write a Line from Memory to a File or Device

This command writes an ASCII line to a file opened on a specified channel. Writing stops when the system hits a carriage return, form feed, or null. The system writes the line to the beginning of a disk file if you .OPENed it. If you opened it via .APPEND, the system appends the line to material already in the file.

Format:

```
.SYSTM
.WRL n    ; Write to channel n.
error return
normal return
```

Require input:

AC0 - byte pointer to memory area from which the line will be read.

Return:

AC1 - byte count of characters written, including terminator.

AC2 - error code.

Example:

This example writes the line read from the console by .RDL to file "OUTPUT".

```
        .
        .
LDA 0, SPACE    ;SPACE WAS DEFINED
.SYSTM          ;EARLIER.
.WRL 1          ;WRITE TO FILE "OUTPUT"
                ;ON CHANNEL 1.
JMP ER
        .
        .
        .
```

.RTN or .ERTN

Return to the Program Above

RDOS and DOS offer five levels of program operation; the CLI normally operates on level 0, and any program you execute from the CLI executes on level 1.

By ending your program with .RTN (or .ERTN), you'll return control to the CLI when the program has executed. If you omitted .RTN (or .ERTN), you'd need to hit an interrupt like CTRL-A to return the CLI, but this lacks finesse.

Call .RTN simply returns control to the CLI, while .ERTN passes the contents of AC2 to the CLI. On a system call error, the system places an identifying error number in AC2. So, if a program issues .ERTN when a call error occurs, the system will pass the error code to the CLI; the CLI will interpret the number returned in AC2 and type an explanatory message on the console. Thus .ERTN is generally useful as an error handler, as shown in the .CRAND example.

Format:

```
.SYSTM
.RTN (or .ERTN)
error return
```

Required input:

None.

Example (to terminate our little program):

```
      .
      .
.SYSTM
.RTN
JMP ER
```

Common Errors

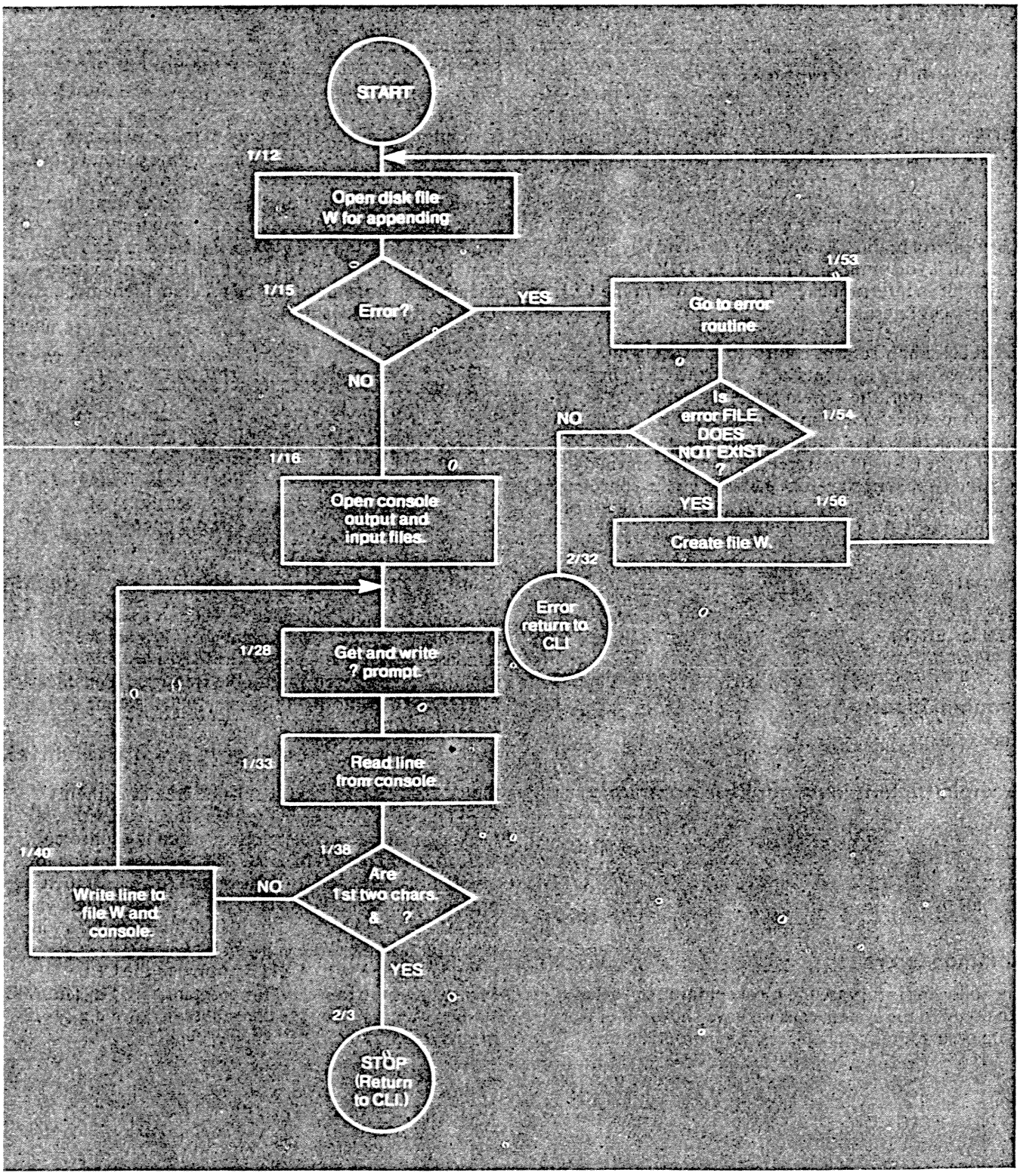
The following errors are the most common ones that can occur on the system calls we have shown. If one of them occurs, the system places the appropriate octal error number in AC2, then goes to the error return. You can have an error processing routine examine this number and act on what it finds. The "mnemonic" shown next to the error number means the same thing as the number; it is defined in the system parameter file, PARU.SR, and you can use it instead of the number if you insert PARU/S (PARU/Scan) in the assembler command before the program filename; e.g., ASM PARU/S filename).

AC2	Mnemonic	Error on call was:
0	ERFNO	Illegal channel number.
1	ERFNM	Illegal filename.
6	EREOF	End of file.
11	ERCRE	File already exists.
12	ERDLE	File does not exist.
15	ERFOP	File not open.

Example Program

The example program, called WRITE, is a short assembly language program that goes through the standard I/O cycle and uses three files: a disk file named W, the console output file (reserved filename STTO) and console input file (reserved filename STTI).

If disk file W doesn't exist, WRITE creates it; then WRITE opens all three files and prompts you with a question mark. It accepts a line typed on the console, and, after you press) to end the line, writes the line to disk file W and echoes the line on the console. It then prompts you again. WRITE opens W for appending via call .APPEND, which means that all lines you type will accumulate in file W. When you type a special terminating line (&)), WRITE returns to the CLI. Figure 9-2 is the flowchart for program WRITE. The slashed numbers are page/line abbreviations; e.g., 1/12 means page 1, line 12.



SD-01630

Figure 9-2. WRITE Flowchart

Writing and Assembling WRITE.SR

The following pages show WRITE and analyze it line by line. They show listing lines as page/line; e.g., 1/6 means page 1, line 6. You might want to examine WRITE in Figure 9-3 before reading the next paragraph.

If you want to try WRITE, you'll need a number of utility files, so begin in the master directory. Use Superedit to create a file named WRITE.SR and type in all the instructions shown in the program listing in Figure 9-3. This is an assembler 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 space before .ER: down to the space before .ERDL on the first page of the listing) and use this as a guide.

If you have an upper- and lowercase console, you can code in either. The most legible way is to put source code in CAPITALS and comments and text strings in upper- and lowercase. The assemblers don't care what case you use, but legibility is important.

When you've finished typing in WRITE, close the file (UE$H$$).

Assembling WRITE

You can choose between the ASM and MAC asemblers for WRITE. We have used MAC for the listing shown, but either one will work.

To use ASM, type:

```
ASM/X PARU/S WRITE $LPT/L)
```

or, if you don't have a line printer, type:

```
ASM/X PARU/S WRITE $TTO/L)
```

If your printing terminal is connected to the second teletypewriter interface, type $TTO1/L instead of $TTO/L. From either ASM command, you should get a listing like the one in Figure 9-3, but without the MACRO header. Although the paging will differ a little, the critical locations are the same.

If, as we did, you want to use the MAC assembler, you may need to build the permanent symbol file for it. Generally, this is needed only once; MAC then uses it for all future assemblies. Try typing:

```
MAC PARU/S WRITE)
```

If you get U or F errors on instructions like LDA or .SYSTM, then you must build a new MAC.PS. This is easy to do.

If you have a NOVA 4 computer, type:

```
MAC/S NBID OSID NSID N4ID PARU)
```

For a NOVA 3, type:

```
MAC/S NBID OSID NSID PARU)
```

For a microNOVA, type:

```
MAC/S MBID OSID NSID PARU)
```

For other NOVA computers, type:

```
MAC/S NBID OSID PARU)
```

For ECLIPSE machines, type:

```
MAC/S NBID OSID NEID PARU)
```

The proper command creates MAC.PS for your machine; you need not create MAC.PS again for this book. Other source (-.SR) files you may eventually need for the Macroassembler are described on the Release Notice supplied with your system.

Having checked (or built) MAC.PS, type:

```
MAC WRITE $LPT/L)
```

or, lacking a line printer, type:

```
MAC WRITE $TTO/L)
```

for a console listing. If your printing terminal is connected to the second teletypewriter interface, type $TTO1/L instead of $TTO/L. From either MAC command, you should get a listing like the one in Figure 9-3, plus the cross-reference.

```
 0001 WRITE    MACRO REV 06.30          16:03:07 04/02/79
                       .TITL WRITE      ; Title is optional but sometimes useful.
 GU                    .ENT  START      ; Global entry for debugging.
03       000001        .TXTM 1          ; Pack text bytes left to right.
04
05                     .ZREL            ; Pointer in page zero avoids assembler A error.
06 00000-000175'.ER:   ERROR            ; .ER contains "ERROR" address in NREL space.
07
08                     .NREL            ; Now for the code proper.
09
10              ; Open the three files.
11
12 00000'020454 .START: LDA 0, NW       ; Point to disk filename W.
13 00001'006017         .SYSTM          ; System,
14 00002'012400         .APPEND 0       ; open file W for append on channel 0,
15 00003'000436         JMP  NOFIL      ;   Error on APPEND, go to NOFIL routine.
  U00004'020000         LDA 0, NTTO     ; Point to console output filename.
17 00005'126400         SUB 1, 1        ; To open a device, use default disable mask.
18 00006'006017         .SYSTM          ; System,
19 00007'014001         .OPEN 1         ; open console output on channel 1,
20 00010'002000-        JMP @ .ER       ;   Error return via ZREL pointer to CLI.
21 00011'020451         LDA 0, NTTI     ; Point to console input (keyboard) filename.
22                                      ; Default mask:  AC1 still = 0.
23 00012'006017         .SYSTM          ; System,
24 00013'014002         .OPEN  2        ; open console keyboard on channel 2.
25
26              ; Begin the main program loop.
27
28 00014'020452 LOOP:   LDA 0,PROMT     ; Point to prompt character.
29 00015'006017         .SYSTM          ; System,
30 00016'017001         .WRL 1          ; write prompt to channel 1 (console out).
31 00017'002000-        JMP @ .ER       ;   Error - report via CLI.
32 00020'020451         LDA 0, SPACP    ; Point to beginning of line buffer.
33 00021'006017         .SYSTM          ; System,
34 00022'015402         .RDL 2          ; read a line from channel 2 (console keyboard).
35 00023'002000-        JMP @ .ER       ;   Error - report via CLI.
36 00024'030446         LDA 2, SPACE    ; Get 1st word (2 bytes) of line buffer in AC2.
37 00025'024412         LDA 1, TWORD    ; Put terminator word (&^) in AC1.
38 00026'132405         SUB 1, 2, SNR   ; Were 1st 2 bytes  &^ ? (Skip if not.)
39 00027'000422         JMP TOCLI       ;   Yes -- return to CLI.
40 00030'006017         .SYSTM          ; No -- System,
41 00031'017000         .WRL 0          ; write line to channel 0, file W.
42 00032'002000-        JMP @ .ER       ;   Error -- report.
43 00033'006017         .SYSTM          ; System,
44 00034'017001         .WRL 1          ; echo the line on console out.
45 00035'002000-        JMP @ .ER       ;   Error -- report.
46 00036'000756         JMP LOOP        ; Now go it all again.
47 00037'023015 TWORD:  .TXT "&<15>"    ; Terminator word, & in left byte, CR in right.
48       000000
49
50              ; Error processing routine, creates file W if it doesn't exist.  When
51              ; system gets here, AC0 still has W pointer, AC2 has error number.
52
53 00041'024407 NOFIL:  LDA 1, .ERDL    ; Get code ERDLE (file does not exist).
54 00042'146404         SUB 2, 1, SZR   ; Skip next instruction if file already exists.
55 00043'002000-        JMP  @ .ER      ;   Error other than ERDLE, exit.
56 00044'006017         .SYSTM          ; System,
57 00045'007000         .CRAND          ; create file W.
58 00046'002000-        JMP @ .ER       ;   Error, exit and report.
  U00047'000000         JMP START       ; Now go back and open the new file.
60 00050'000012 .ERDL:  ERDLE           ; ERDLE = 12.
```

Figure 9-3. WRITE.SR Program with Errors

```
 0002 WRITE
01                ;Normal return to CLI.
02
03 00051'006017 TOCLI:  .SYSTM            ; System,
04 00052'004400         .RTN              ; return to CLI.
05 00053'002000-        JMP  @ .ER        ;   Error (can't happen).
06
07                ; Filenames, prompt, buffer, etc.
08
09 00054'000132"NW:     .+1*2             ; Point to filename W.
10 00055'053400         .TXT "W"          ;
11
 FU00056'000000 NTTO    .+1*2             ; Point to filename $TTO.
13 00057'022124         .TXT "STT0"     ;
14       052060
15       000000
16 00062'000146"NTTI:   .+1*2             ; Point to filename $TTI.
17 00063'022124         .TXT  "STTI"      ;
18       052111
19       000000
20
21 00066'000156"PROMT:   .+1*2            ; Point to prompt
22 00067'037415         .TXT "?<15>"      ;  "?" and CR (#).
23       000000
24
25 00071'000164"SPACP:  SPACE*2           ; Point (addr*2) to 1st byte in line buffer.
26
27 00072'000103 SPACE:  .BLK 132./2+1     ; 133. bytes for read/write line buffer-
28                                        ; let the assembler compute it.
29
30                ;Error return to CLI.
31
32 00175'006017 ERROR:  .SYSTM            ; System,
33 00176'006400         .ERTN             ; return to CLI and have CLI report.
34 00177'002000-        JMP  @ .ER        ;   Error (can't happen).
35
 UU                     .END START

**00008 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

Figure 9-3. WRITE.SR Program with Errors (continued)

The program .ENTers its starting address, START, not because other modules will use it, but because we want to identify START symbolically to the debugger. Later this will help us debug the program. The .ENT line is flagged with a GU error.

The .ZREL pointer on line 1/6 points to error handler ERROR at the end of page 2. Through this pointer, any location in 32K words can get to ERROR. Without the pointer, ERROR would be nearly out of range of the JMP in line 1/20. All system calls except the first, .APPEND, make use of this .ER pointer for their error returns.

The .NREL in line 8 specifies normal relocation, which applies to the rest of the program.

The first group of code, in lines 1/12-1/24, opens disk file W for appending and opens the console output and input files for normal I/O (line 16 has a U error). If .APPEND cannot open file, it takes the error return on 1/15; this jumps to the routine on lines 1/52-1/60. This routine creates file W if it doesn't exist, then jumps back to the beginning of the program.

The next block of code starts with LOOP and extends from line 1/28 to line 1/48. This code reads a line typed on the console, and checks to see if the first two characters are & and). It does this by loading the first two words (two bytes or two characters) from line 2/25 and comparing this to a match word (1/47) set up to contain &). If they match, the program returns to the CLI via sequence TOCLI, 2/3. If they don't match, the program writes the entire line to disk file W, echoes it on the console, and returns to LOOP.

Writing and Assembling WRITE.SR

The following pages show WRITE and analyze it line by line. They show listing lines as page/line; e.g., 1/6 means page 1, line 6. You might want to examine WRITE in Figure 9-3 before reading the next paragraph.

If you want to try WRITE, you'll need a number of utility files, so begin in the master directory. Use Superedit to create a file named WRITE.SR and type in all the instructions shown in the program listing in Figure 9-3. This is an assembler 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 space before .ER: down to the space before .ERDL on the first page of the listing) and use this as a guide.

If you have an upper- and lowercase console, you can code in either. The most legible way is to put source code in CAPITALS and comments and text strings in upper- and lowercase. The assemblers don't care what case you use, but legibility is important.

When you've finished typing in WRITE, close the file (UE$H$$).

Assembling WRITE

You can choose between the ASM and MAC asemblers for WRITE. We have used MAC for the listing shown, but either one will work.

To use ASM, type:

```
ASM/X PARU/S WRITE $LPT/L)
```

or, if you don't have a line printer, type:

```
ASM/X PARU/S WRITE $TTO/L)
```

If your printing terminal is connected to the second teletypewriter interface, type $TTO1/L instead of $TTO/L. From either ASM command, you should get a listing like the one in Figure 9-3, but without the MACRO header. Although the paging will differ a little, the critical locations are the same.

If, as we did, you want to use the MAC assembler, you may need to build the permanent symbol file for it. Generally, this is needed only once; MAC then uses it for all future assemblies. Try typing:

```
MAC PARU/S WRITE)
```

If you get U or F errors on instructions like LDA or .SYSTM, then you must build a new MAC.PS. This is easy to do.

If you have a NOVA 4 computer, type:

```
MAC/S NBID OSID NSID N4ID PARU)
```

For a NOVA 3, type:

```
MAC/S NBID OSID NSID PARU)
```

For a microNOVA, type:

```
MAC/S MBID OSID NSID PARU)
```

For other NOVA computers, type:

```
MAC/S NBID OSID PARU)
```

For ECLIPSE machines, type:

```
MAC/S NBID OSID NEID PARU)
```

The proper command creates MAC.PS for your machine; you need not create MAC.PS again for this book. Other source (-.SR) files you may eventually need for the Macroassembler are described on the Release Notice supplied with your system.

Having checked (or built) MAC.PS, type:

```
MAC WRITE $LPT/L)
```

or, lacking a line printer, type:

```
MAC WRITE $TTO/L)
```

for a console listing. If your printing terminal is connected to the second teletypewriter interface, type $TTO1/L instead of $TTO/L. From either MAC command, you should get a listing like the one in Figure 9-3, plus the cross-reference.

```
 0001 WRITE    MACRO REV 06.30          16:03:07 04/02/79
                        .TITL WRITE     ; Title is optional but sometimes useful.
 GU                     .ENT  START     ; Global entry for debugging.
03       000001         .TXTM 1         ; Pack text bytes left to right.
04
05                      .ZREL           ; Pointer in page zero avoids assembler A error.
06 00000-000175'.ER:    ERROR           ; .ER contains "ERROR" address in NREL space.
07
08                      .NREL           ; Now for the code proper.
09
10              ; Open the three files.
11
12 00000'020454 .START: LDA 0, NW       ; Point to disk filename W.
13 00001'006017         .SYSTM          ; System,
14 00002'012400         .APPEND 0       ; open file W for append on channel 0,
15 00003'000436         JMP  NOFIL      ;   Error on APPEND, go to NOFIL routine.
  U00004'020000         LDA 0, NTTO     ; Point to console output filename.
17 00005'126400         SUB 1, 1        ; To open a device, use default disable mask.
18 00006'006017         .SYSTM          ; System,
19 00007'014001         .OPEN 1         ; open console output on channel 1,
20 00010'002000-        JMP @ .ER       ;   Error return via ZREL pointer to CLI.
21 00011'020451         LDA 0, NTTI     ; Point to console input (keyboard) filename.
22                                      ; Default mask:  AC1 still = 0.
23 00012'006017         .SYSTM          ; System,
24 00013'014002         .OPEN  2        ; open console keyboard on channel 2.
25
26              ; Begin the main program loop.
27
28 00014'020452 LOOP:   LDA 0,PROMT     ; Point to prompt character.
29 00015'006017         .SYSTM          ; System,
30 00016'017001         .WRL 1          ; write prompt to channel 1 (console out).
31 00017'002000-        JMP @ .ER       ;   Error = report via CLI.
32 00020'020451         LDA 0, SPACP    ; Point to beginning of line buffer.
33 00021'006017         .SYSTM          ; System,
34 00022'015402         .RDL 2          ; read a line from channel 2 (console keyboard).
35 00023'002000-        JMP @ .ER       ;   Error = report via CLI.
36 00024'030446         LDA 2, SPACE    ; Get 1st word (2 bytes) of line buffer in AC2.
37 00025'024412         LDA 1, TWORD    ; Put terminator word (&↵) in AC1.
38 00026'132405         SUB 1, 2, SNR   ; Were 1st 2 bytes  &↵ ? (Skip if not.)
39 00027'000422         JMP TOCLI       ;   Yes -- return to CLI.
40 00030'006017         .SYSTM          ; No -- System,
41 00031'017000         .WRL 0          ; write line to channel 0, file W.
42 00032'002000-        JMP @ .ER       ;   Error -- report.
43 00033'006017         .SYSTM          ; System,
44 00034'017001         .WRL 1          ; echo the line on console out.
45 00035'002000-        JMP @ .ER       ;   Error -- report.
46 00036'000756         JMP LOOP        ; Now do it all again.
47 00037'023015 TWORD:  .TXT "&<15>"    ; Terminator word, & in left byte, CR in right.
48       000000
49
50              ; Error processing routine, creates file W if it doesn't exist.  When
51              ; system gets here, AC0 still has W pointer, AC2 has error number.
52
53 00041'024407 NOFIL:  LDA 1, .ERDL    ; Get code ERDLE (file does not exist).
54 00042'146404         SUB 2, 1, SZR   ; Skip next instruction if file already exists.
55 00043'002000-        JMP  @ .ER      ;   Error other than ERDLE, exit.
56 00044'006017         .SYSTM          ; System,
57 00045'007000         .CRAND          ; create file W.
58 00046'002000-        JMP @ .ER       ;   Error, exit and report.
  U00047'000000         JMP START       ; Now go back and open the new file.
60 00050'000012 .ERDL:  ERDLE           ; ERDLE = 12.
```

Figure 9-3. WRITE.SR Program with Errors

```
 0002 WRITE
01                ;Normal return to CLI.
02
03 00051'006017 TOCLI:  .SYSTM            ; System,
04 00052'004400         .RTN              ; return to CLI.
05 00053'002000-        JMP  @ .ER        ;   Error (can't happen).
06
07                ; Filenames, prompt, buffer, etc.
08
09 00054'000132"NW:     .+1*2             ; Point to filename W.
10 00055'053400         .TXT "W"          ;
11
 FU00056'000000 NTTO    .+1*2             ; Point to filename STTO.
13 00057'022124         .TXT "STT0"     ;
14       052060
15       000000
16 00062'000146"NTTI:   .+1*2             ; Point to filename STTI.
17 00063'022124         .TXT  "STTI"      ;
18       052111
19       000000
20
21 00066'000156"PROMT:   .+1*2            ; Point to prompt
22 00067'037415         .TXT "?<15>"      ;  "?" and CR (V).
23       000000
24
25 00071'000164"SPACP:  SPACE*2           ; Point (addr*2) to 1st byte in line buffer.
26
27 00072'000103 SPACE:  .BLK 132./2+1     ; 133. bytes for read/write line buffer-
28                                        ; let the assembler compute it.
29
30                ;Error return to CLI.
31
32 00175'006017 ERROR:  .SYSTM            ; System,
33 00176'006400         .ERTN             ; return to CLI and have CLI report.
34 00177'002000-        JMP  @ .ER        ;   Error (can't happen).
35
 UU                     .END START

**00008 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

Figure 9-3. WRITE.SR Program with Errors (continued)

The program .ENTers its starting address, START, not because other modules will use it, but because we want to identify START symbolically to the debugger. Later this will help us debug the program. The .ENT line is flagged with a GU error.

The .ZREL pointer on line 1/6 points to error handler ERROR at the end of page 2. Through this pointer, any location in 32K words can get to ERROR. Without the pointer, ERROR would be nearly out of range of the JMP in line 1/20. All system calls except the first, .APPEND, make use of this .ER pointer for their error returns.

The .NREL in line 8 specifies normal relocation, which applies to the rest of the program.

The first group of code, in lines 1/12-1/24, opens disk file W for appending and opens the console output and input files for normal I/O (line 16 has a U error). If .APPEND cannot open file, it takes the error return on 1/15; this jumps to the routine on lines 1/52-1/60. This routine creates file W if it doesn't exist, then jumps back to the beginning of the program.

The next block of code starts with LOOP and extends from line 1/28 to line 1/48. This code reads a line typed on the console, and checks to see if the first two characters are & and). It does this by loading the first two words (two bytes or two characters) from line 2/25 and comparing this to a match word (1/47) set up to contain &). If they match, the program returns to the CLI via sequence TOCLI, 2/3. If they don't match, the program writes the entire line to disk file W, echoes it on the console, and returns to LOOP.

On most errors, the program jumps through ZREL to ERROR, near the end of page 2. ERROR returns via .ERTN, which tells the CLI to interpret the error.

Now for the other locations. We need a byte pointer and text string for filename W, STTO, STTI, and for the prompt (? and), ASCII 15). Lines 2/9 through 2/23 contain these byte pointers and text strings. SPACP, line 2/25, is a byte pointer to the start of the line buffer, needed for read and write line calls. It is separate from the buffer name because we need a unique symbol to load the first word of the buffer. Line 2/26 establishes the buffer itself; the read call reads into this and the write call writes from it.

Finally, we .END the program and direct control back to START. The .END statement is flagged with two U errors.

The MAC listing shows 8 errors, on lines 1/2, 1/16, 1/59, and 2/26. The ASM listing shows fewer errors, but in the same locations. As with error reports from 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 error conditions. The first error results from our use of the symbol .START. We intended to use symbol START as a name for the beginning of the program. Deleting the period from the symbol should clear up several errors.

The second error clearly relates to symbol NTTO since all lines which contain NTTO are flagged. The problem lies in line 2/12 where we forgot to insert a colon after NTTO. Insert a colon, creating the valid label NTTO: and other errors should go away.

WRITE.SR with No Assembly Errors

Having corrected the seven assembly errors with Superedit, reassemble the program with the command:

{ASM/X PARU/S / MAC} WRITE {$LPT/L / STTO/L / STTO1/L})

From this, you get a listing and an RB file. The listing shows no errors. You're now ready to process WRITE with RLDR. Type:

```
RLDR/D WRITE)
```

The global /D switch loads the debugger with WRITE, so that you can debug it later.

RLDR types out a listing of modules loaded with the program, some NMAX and ZMAX figures, but no errors. Now to execute WRITE, type:

```
WRITE)
FILE DOES NOT EXIST:WRITE.SV
R
```

Curious. Not only did WRITE bomb without typing its prompt, but stated that it didn't exist. Actually, this is the way .ERTN works: it passes an error code to the CLI, but can't pass anything else because it has only AC2 to work with. The error must have been code 12_8, which the CLI correctly interpreted as *FILE DOES NOT EXIST*. The ": WRITE.SV" appeared because the CLI always tries to identify the argument that caused the error. Here it failed because WRITE had replaced it in memory. When it returned, all it knew was that a program named WRITE.SV had issued .ERTN with 12 in AC2. Thus, the filename argument that the CLI gives after a program issues .ERTN is always the program name itself, and you can ignore it.

The CLI's *explanation* of the error *(FILE DOES NOT EXIST)* is accurate, though. One of WRITE'S system calls took the error return because it could not find a file. This particular call must be one of the first three calls because WRITE didn't type the prompt on the console. It couldn't have been the .APPEND call, because .APPEND will jump to a file-creating routine if its file doesn't exist. Therefore it must be one of the console-opening .OPEN calls. The calls themselves look ok, as do the byte pointers (one of whose labels you fixed) on lines 2/12 and 2/16. The console device names on lines 2/13 and 2/17 also seem ok -- but no! Line 2/13 has the console output name as STT0. The correct name is STTO. This explains the error: WRITE tried to open file STT0, which didn't exist.

Using Superedit, change STT0 to STTO in WRITE.SR. Then assemble the program again as shown earlier and run it through RLDR again. The assembler will delete the old, defective RB binary file and replace it with the new one; RLDR will do the same with the save file.

Run the new WRITE.SV again:

```
WRITE)
STTI
```

At least it didn't bomb. "STTI" may not be the correct prompt, but it's an improvement over "FILE DOES NOT EXIST". Type)

```
)
?
```

The *question mark* is the right prompt. Type something:

```
SOMETHING)
SOMETHING
?
```

WRITE repeated the line on the console -- so far so good -- but we don't know whether it wrote to the disk file yet. The prompt problem seems to have fixed itself.

```
SOMETHING ELSE)
SOMETHING ELSE
?
```

To check the disk file, you'll need to get back to the CLI. Try the terminating sequence (&):

```
&)
R
```

The terminator works and you're back to the CLI. Now what about the disk file, W?

```
TYPE W)
SOMETHING
SOMETHING ELSE
R
```

The disk file mechanism seems fine, too. Aside from the initial prompt problem, WRITE seems to be in pretty good shape. Check by running it again:

```
WRITE)
STTI )

?
SIGH)
SIGH
?
```

Clearly the problem isn't going to go away. Stop WRITE and check file W from the CLI:

```
&)
R
TYPE W)
SOMETHING
SOMETHING ELSE

SIGH
R
```

Everything works except the inital prompt, which is $TTI when it should be ?. The next step is a very common one in assembly language programming: debugging.

Introducing the Debugger

This section gives a debugger overview. Read it before you proceed to operate the debugger, as described in the next main section, Debugging Write.

The debugger is a system utility that lets you examine and make minor modifications to examine your program as you run sections of it. There are two debuggers, IDEB, which allows you to disable device interrupts, and DEB, which does not. This section deals with DEB. You'll need IDEB only to debug routines that handle nonstandard device interrupts. For more on either debugger, consult the *Symbolic Debugger* manual.

Remember that we loaded DEB with WRITE, assuming that we might need it later. When we've found the bug and corrected it, we can reload WRITE without DEB.

Most debug commands employ the ESC key, which, as in SUPEREDIT, echoes as "$". Here the DEB commands you'll be using:

Command	Meaning
[adr]/	Display contents of *adr*.
[ac]$A	Display contents of one or all accumulators.
[debug-adr] $B	Insert a breakpoint at *debug-adr*, or display all breakpoints.
[n] $D	Delete one or all breakpoints.
$P	Proceed to execute the program from the current breakpoint.
[adr] $R	Run the program from the beginning, or from *adr*. You must issue $R initially; then you can use $P, or adr$R.
$V or CTRL-A	Leave the debugger and return to the CLI.
=`;?	These commands control the format of debugger display; we describe them below.
NEW LINE/ LINE FEED UPARROW (↑)	These tell the debugger to proceed or back up one location.

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 contents of the accumulators and to look at whatever else you want in your program. You may setup to eight breakpoints in a program. Each breakpoint address you specify can be either an absolute number or a symbolic expression. Symbolic expressions are easier. You can use each location you .ENTered in a program to specify a breakpoint address; e.g., LOOP+3 or ER+1, if you had .ENTered LOOP and ER. By entering START in WRITE, named a symbolic location that you can use in debugger address expressions.

Changing Display Format

Initially, the debugger displays the contents of locations in octal; this is the default mode. You can change it within a command line by typing one of the following local commands (we don't show them all here):

Command	Meaning
=	display last item in octal.
`	display last item in ASCII.
;	display last item in instruction format (e.g., LDA 0, OUTPUT)
?	display last item in .SYSTM call format (e.g., .WRL 0)
&	display last item in byte pointer format (e.g., 000522 0)

You can change the entire output mode of the debugger by typing ESC, then one of these commands; the new mode will remain until you change it. For example, $` tells the debugger to display everything in ASCII.

The debugger doesn't understand DEL or RUBOUT, nor does it accept ESCape commands in the middle of a line. If you type something it doesn't like, it will display a "U" or question mark (?), then output a return and wait for another command.

Examining and Changing Memory Locations

You can display the contents of any location by typing the symbolic or numberic address of the location followed by a slash. For example,

START/ *020454*

displays the contents of START. START is a *debug address*. (Remember that we show user input in **boldface** type and system output in *italic* type.) To display successively higher locations, press the NEW LINE key, or if your terminal doesn't have a NEW LINE key, the LINE FEED key. We show this key as downarrow (↓) although the debugger doesn't echo it at all on the terminal. To display successively lower locations, type uparrows via the SHIFT-6 or SHIFT-N key. We show uparrows as (↑), as does the debugger. For example:

```
                  !
START+1     006017↓
START+2     012400↑
START+1     006017
```

Octal numbers by themselves don't mean very much. You can tell the debugger to interpret them by typing one or more local display commands.

For example, address START contains an octal number:

START/ *020454*

Semicolon (;) displays this number in instruction format:

;*LDA START+54*

Question mark (?) displays it in .SYSTM call format:

?.*CCON 54*

Apostrophe (') displays it in ASCII format:

'*!*

and finally, ampersand (&) displays it in byte-pointer format:

&*010226 0*

For any location, only one local display command gives a useful picture of the contents. If a locations holds an instruction, like JMP LDA, the appropriate command is ";", which displays in instruction format. The assembler listing can help in this process; for example, the listing in Figure 9-3 tells you that START contains an instruction, thus the semicolon is the appropriate display command.

If a location holds a system call, like .OPEN 1 or .RDL 2, the appropriate command is "?", which displays in .SYSTM call format. If a location holds ASCII characters, like ST or W, the appropriate command is "'", which gives ASCII format. For a byte pointer, like ST, the command is &, followed by a slash, which displays the contents of the location pointed to. Usually, you can tell which kind of display command is correct because other commands will give absurd results, like .CCON 54 and !, and 010226 0 above.

To change the contents of memory location, display the location with a / or ↓ or ↑ command, then type in the contents you want and press). The new contents can be an octal number or it can be an instruction like LDA 0, START+55. A word of caution about changing locations: the computer will execute the new contents without asking questions, so problems could occur if you make a mistake. Also, be wary of changing a location's contents accidentally. If, when a location is open, after / or ↓, or ↑, you inadvertently type something and press), the character(s) you typed may replace the old contents of the location. These new contents may cause problems when you run the program. If you suspect that you have accidentally modified a location, and haven't yet pressed), press the DEL or RUBOUT key. The debugger will reject the entire line and then will type ? or U; you can then proceed. If you *have* pressed) after accidentally changing something, type CTRL-A to get back to the CLI and type the debug command again; this will bring the original program (which remains unchanged on disk) back into memory.

Starting or Continuing to Run Your Program

To run your program for the first time, type $R; thereafter you can proceed with program execution by typing P or addr$R. For 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.

Ending a Debugging Session

When you are done debugging and want to return to the CLI, simply type

$V (or CTRL-A)

The changes you have made to your program during the debugging session do not become part of the disk file when you leave the debugger. To make permanent changes to your program you must go through the cycle of editing your source program, assembling it, and processing it with RLDR.

Debugging WRITE

You've discovered and eliminated the assembly and filename errors from WRITE. It now runs properly, except for its first pass, when it displays $TTI instead of ? for a prompt.

Now, having reviewed debugger fundamentals, you can use the debugger to find out why WRITE gives the erroneous prompt.

Type

DEB WRITE)

to activate the debugger, which responds with a carriage return. The carriage return is its prompt, like Superedit's !.

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 .WRL in line 1/30 (Figure 9-3) gives $TTI as an initial prompt, instead of the ? that line 1/28 specifies. After the program runs through the loop once, it gives the correct prompt. The problem must be somewhere around 1/28, because the LDA isn't working on the first pass.

For practice, take a look at the locations around START. Start with START:

START/ *020454* ; *LDA 0 START+54*

Press ↓ to open and display the next location:

↓

START+1 006071 ; *JSR @ +17*

This is what the .SYSTM mnemonic before each system call looks like in memory. It is a JSR to location 17_8, from which location the system processes the system call.

↓

START+2 012400 ; *ISZ START+2 ? .APPE 0*

There's the .APPEND system call word. The debugger displays only five characters. Without the preceeding .SYSTM word, the computer would have executed the 012400 as the ISZ instruction -- which is *not* what you want.

↓

START+3 000436 ; *JMP START+41*

This is the error return from .APPEND, which goes to NOFIL on line 1/53. Try backing up a location:

↑

START+2 012400

Jump around a bit:

)
START+100/ *000000* ; *JMP 0*)

This is an empty word in the line buffer, not a JMP 0 instruction.

START+37/ *023015* ; *LDA 0 +152 '& < 15 >*)

This is the matching TWORD at line 1/47.

Having looked around, try setting a breakpoint at START.
Type:

START$B

This sets a breakpoint at the location immediately *preceding* START. Display the breakpoints:

$B
7B START

The debugger names the first breakpoint 7B, the second 6B, and so on, to the eighth breakpoint, 0B.

Run the program:

$R
7B START
0 000000 1 000000 2 000000 3 000000

When control reached the breakpoint, it went to the debugger, which displayed the breakpoint name, 7B, location, START, and the contents of the accumulators. AC0, AC1, AC2, and AC3 and each contain 0 since the program hasn't loaded anything into them yet.

Set another breakpoint at the next location, START+1:

START+1$B

After you have type $R once, you use $P to proceed with the program. You can use $R again only if you precede it with an address, like START. Type:

```
$P
6B START+1
0 001244  1  000000  2  000000  3  000000
```

Something got into AC0 -- looks like a byte pointer. You can check by displaying AC0's contents in byte-pointer format, then displaying the byte pointer location's contents in ACSII:

```
OSA  001244  &  000522 0  )
```

AC0 contains 1244, which, in byte-pointer format, is 522. Now what does 522 contain in ASCII?

```
552/  053400  '  W < 0 >
```

It contains W for the disk filename -- as it should.

Return to START+2 and move forward to the next LDA instruction:

```
START+2/  012400  ?.APPE 0 ↓
START+3  000436 ; JMP START+41  ↓
START+4  020452  ;  LDA 0 START+56  ↓
START+5  126400  ;  SUB 1 1
```

This looks like a good place -- after the LDA instruction in START+4. Set another breakpoint and proceed:

```
)
START+5$B
$P
5B START+5
0 001250  1  00000  2  000000  3  000000
```

Another byte pointer in AC0; check it out:

```
OSA 001250 & 000524 0 524/ 022124 ' ST
```

AC0 contains a pointer to a text string beginning with ST. The LDA in START+4 worked and everything is ok thus far.

Now, get to LOOP and see if the byte pointer to the ? prompt gets loaded:

```
)
START+13/ 014002 ; DSZ +2 ?.OPEN 2 ↓
START+14  020452  ; LDA 0 START+66
```

Set another breakpoint after the LDA in START+14:

```
)
START+15$B
$P
4B START+15
0  001260  1  000000  2  000000  3  000000
```

AC0 contains 1260, which should be a bytepointer to the prompt (? < 15 >). Check it out:

```
OSA 001260 & 000530 0 530/ 022124 ' ST
```

The "ST" doesn't look much like the prompt, but try the next location:

```
START+64 052111  ' TI
```

AC0 still has the byte pointer to STTI, not to ? < 15 >, as it should. This explains why WRITE says STTI instead of ?. It means that the LDA in location LOOP isn't happening. Why not? Continue the program:

```
)
$P STTI
```

This is WRITE's .WRL at line 1/30. Type some text next to the STTI prompt:

```
    FOO)
FOO
4B START+15
0 001270  1  000004  2  027070  3  000000
```

WRITE has jumped back to LOOP and is executing the LOOP breakpoint again. AC1 shows the byte count, including the CR (carriage return), from the .WRL on 1/44; the byte counts from .RDLs don't include CRs. AC2 shows the results of the SUB on line 1/38.

But AC0 differs from the first time you saw this breakpoint. It probably has the right byte pointer now. Check it:

```
OSA 001270 & 000534 0 534/ 037415 ' ? < 15 >  )
```

Yes. This is the right prompt. WRITE will display it when you proceed:

```
$P ?
```

? is the correct prompt. Try typing something else:

```
ZUT ALORS)
ZUT ALORS

4B START+15
0  001270  1  000012  2  032110  3  000000
```

The byte pointer in AC0 remains the same, which is consistent with your earlier experience: WRITE gave the correct prompt every time after the first pass. Clearly the loop is ok now.

The debug session has shown that WRITE skips the LDA in line LOOP for its first pass, but executes this LDA correctly thereafter. Now why does it do that?

Because we forgot the error return from the .SYSTM call that opens the console input file, line 1/25.

The error return and the .SYSTM word are essential parts of each system call, but people too often forget them. In this case, the system opens file $TTI normally, but assumes that the LDA in line LOOP is the error return, thus the system skips it. When WRITE gets to the .WRL for the prompt, AC0 still contains a byte pointer to $TTI, which is why you get $TTI 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:

```
CTRL-A
INT
R
DEB WRITE)
```

Set the critical breakpoint and run the program:

```
START+15$B
$R
7B START+15
0  001260  1  000000  2  00000  3  000000
```

Correct the byte pointer in AC0:

```
OSA 001260 1270)
```

Verify the change:

```
OSA 001270 & 000534 0 534/ 037415 ' ? < 15 > )
```

Looks good. Now when you proceed, WRITE should say ?), not $TTI:

```
$P ?
```

Solid. The right prompt on the first pass. The problem is definitely the missing error return, which you must insert with Superedit. Get back to the CLI:

```
&)
R
```

If you check file W (TYPE W)), you'll find that the test lines typed during the debugging process were appended to it.

Final Version of WRITE

Get into WRITE.SR with Superedit, and insert a JMP @ .ER instruction after the .OPEN 2 call in line 1/24. Don't insert) (CR) if you want the assembler listing line numbers to remain the same. Simply type something like:

```
!J$SNTTI$S@$T$S
    .OPEN 2(↑) ; open console keyboard...
!S)
$T$S
!ItabJMP @ .ER$-2L$4T$S
          .SYSTEM   ; System
          .OPEN 2   ; open file W...
          JMP @ .ER
; Begin the main program loop.
!
```

Then close the file (UESHSS), and reassemble WRITE. Run it through RLDR again, *without* the global /D switch:

```
RLDR WRITE  { SLPT/L }
            { STTO/L }
            { $TTO1/L } )
```

Figure 9-4 shows the final version of WRITE, including the cross reference. Shaded areas indicate changes from the initial version.

```
 0001 WRITE    MACRO REV 06.30          17:58:13 04/02/79
                        .TITL WRITE     ; Title is optional but sometimes useful.
 02                     .ENT  START     ; Global entry for debugging.
 03       000001        .TXTM 1         ; Pack text bytes left to right.
 04
 05                     .ZREL           ; Pointer in page zero avoids assembler A error.
 06 00000-000176'.ER:   ERROR           ; .ER contains "ERROR" address in NREL space.
 07
 08                     .NREL           ; Now for the code proper.
 09
 10              ; Open the three files.
 11
 12 00000'020455 START: LDA 0, NW       ; Point to disk filename W.
 13 00001'006017        .SYSTM          ; System,
 14 00002'012400        .APPEND 0       ; open file W for append on channel 0.
 15 00003'000437        JMP  NOFIL      ;   Error on APPEND, go to NOFIL routine.
 16 00004'020453        LDA 0, NTTO     ; Point to console output filename.
 17 00005'126400        SUB 1, 1        ; To open a device, use default disable mask.
 18 00006'006017        .SYSTM          ; System,
 19 00007'014001        .OPEN 1         ; open console output on channel 1.
 20 00010'002000-       JMP @ .ER       ;   Error return via ZREL pointer to CLI.
 21 00011'020452        LDA 0, NTTI     ; Point to console input (keyboard) filename.
 22                                     ; Default mask:  AC1 still = 0.
 23 00012'006017        .SYSTM          ; System,
 24 00013'014002        .OPEN  2        ; open console keyboard on channel 2.
 25 00014'002000-       JMP @ .ER
 26              ; Begin the main program loop.
 27
 28 00015'020452 LOOP:  LDA 0,PROMT     ; Point to prompt character.
 29 00016'006017        .SYSTM          ; System,
 30 00017'017001        .WRL 1          ; write prompt to channel 1 (console out).
 31 00020'002000-       JMP @ .ER       ;   Error - report via CLI.
 32 00021'020451        LDA 0, SPACP    ; Point to beginning of line buffer.
 33 00022'006017        .SYSTM          ; System,
 34 00023'015402        .RDL 2          ; read a line from channel 2 (console keyboard).
 35 00024'002000-       JMP @ .ER       ;   Error - report via CLI.
 36 00025'030446        LDA 2, SPACE    ; Get 1st word (2 bytes) of line buffer in AC2.
 37 00026'024412        LDA 1, TWORD    ; Put terminator word (&↵) in AC1.
 38 00027'132405        SUB 1, 2, SNR   ; Were 1st 2 bytes  &↵ ? (Skip if not.)
 39 00030'000422        JMP TOCLI       ;   Yes -- return to CLI.
 40 00031'006017        .SYSTM          ; No -- System,
 41 00032'017000        .WRL 0          ; write line to channel 0, file W.
 42 00033'002000-       JMP @ .ER       ;   Error -- report.
 43 00034'006017        .SYSTM          ; System,
 44 00035'017001        .WRL 1          ; echo the line on console out.
 45 00036'002000-       JMP @ .ER       ;   Error -- report.
 46 00037'000756        JMP LOOP        ; Now do it all again.
 47 00040'023015 TWORD: .TXT "&<15>"    ; Terminator word, & in left byte, CR in right.
 48       000000
 49
 50              ; Error processing routine, creates file W if it doesn't exist.  When
 51              ; system gets here, AC0 still has W pointer, AC2 has error number.
 52
 53 00042'024407 NOFIL: LDA 1, .ERDL    ; Get code ERDLE (file does not exist).
 54 00043'146404        SUB 2, 1, SZR   ; Skip next instruction if file already exists.
 55 00044'002000-       JMP  @ .ER      ;   Error other than ERDLE, exit.
 56 00045'006017        .SYSTM          ; System,
 57 00046'007000        .CRAND          ; create file W.
 58 00047'002000-       JMP @ .ER       ;   Error, exit and report.
 59 00050'000730        JMP START       ; Now go back and open the new file.
 60 00051'000012 .ERDL: ERDLE           ; ERDLE = 12.
```

Figure 9-4. WRITE.SR Program without Errors

```
 0002 WRITE
01                ;Normal return to CLI.
02
03 00052'006017 TOCLI:  .SYSTM           ; System,
04 00053'004400         .RTN             ; return to CLI.
05 00054'002000-        JMP  @ .ER       ;   Error (can't happen).
06
07                ; Filenames, prompt, buffer, etc.
08
09 00055'000134"NW:     .+1*2            ; Point to filename W.
10 00056'053400         .TXT "W"         ;
11
12 00057'000140"NTTO:   .+1*2            ; Point to filename $TTO.
13 00060'022124         .TXT "$TTO"      ;
14       052117
15       000000
16 00063'000150"NTTI:   .+1*2            ; Point to filename $TTI.
17 00064'022124         .TXT  "$TTI"     ;
18       052111
19       000000
20
21 00067'000160"PROMT:   .+1*2           ; Point to prompt
22 00070'037415         .TXT "?<15>"     ;  "?" and CR (↵).
23       000000
24
25 00072'000166"SPACP:  SPACE*2          ; Point (addr*2) to 1st byte in line buffer.
26
27 00073'000103 SPACE:  .BLK 132./2+1    ; 133. bytes for read/write line buffer-
28                                       ; let the assembler compute it.
29
30                ;Error return to CLI.
31
32 00176'006017 ERROR:  .SYSTM           ; System,
33 00177'006400         .ERTN            ; return to CLI and have CLI report.
34 00200'002000-        JMP  @ .ER       ;   Error (can't happen).
35              .
36                      .END START

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS

  0003 WRITE

ERROR 000176'        1/06    2/32
LOOP  000015'        1/28    1/46
NOFIL 000042'        1/15    1/53
NTTI  000063'        1/21    2/16
NTTO  000057'        1/16    2/12
NW    000055'        1/12    2/09
PROMT 000067'        1/28    2/21
SPACE 000073'        1/36    2/25    2/27
SPACP 000072'        1/32    2/25
START 000000' EN     1/02    1/12    1/59    2/36
TOCLI 000052'        1/39    2/03
TWORD 000040'        1/37    1/47
.ER   000000-        1/06    1/20    1/25    1/31    1/35    1/42    1/45
                     1/55    1/58    2/05    2/34
.ERDL 000051'        1/53    1/60
```

Figure 9-4. WRITE.SR Program without Errors (continued)

Figure 9-5 shows RLDR information (called a load map) from WRITE.

The RLDR load map, shown for WRITE in Figure 9-5, shows the module names and program size information. It also gives some module starting addresses, which makes a useful debugging tool, although you didn't use it during this debugging session. In Figure 9-5, WRITE is the .TITL of the module. TMIN is the single-task scheduler. If WRITE were a multitask program, you'd have specified the number of tasks with the local /K switch in the RLDR line, and the RLDR would have loaded the multitask scheduler, TCBMON.

NSAC3 tells the system to copy the contents of location 16_8, the USP or User Stack Pointer, into AC3 after each system or task call. Thus you can use USP, a handy ZREL location, to store the return address from a subroutine. No matter how the subroutine uses AC3, a system or task call will restore it to the value it had on entry. The program can then JMP 0,3 to return from the subroutine. The .ENT pseudo-op in Chapter 8 shows an example of USP usage.

```
WRITE.SV LOADED BY
 RLDR REV 07.10 AT 17:59:02 04/02/79

 WRITE
 TMIN
 NSAC3


   NMAX      000742
   ZMAX      000051
   CSZE      000000
    EST      000000
    SST      000000

   USTAD     000400
   START     000445
   TMIN      000647
   .SAC0     020016
   .SAC1     024016
   .SAC2     030016
   .SAC3     034016
```

Figure 9-5. RLDR Load Map from WRITE

NMAX, the first unused word of NREL memory above WRITE code, is 742_8. ZMAX, the first free word of ZREL memory, is 51_8 because the program uses location 50_8 for the ERROR pointer. CSZE is the FORTRAN COMMON area size; here there is no COMMON. EST is the (bottom) of the symbol table. SST is the start of this table. The symbol table helped you debug WRITE, but you didn't tell RLDR to load it (global /D) with this version of WRITE, hence, figures for it are zero.

USTAD is the starting address of WRITE's User Status Table. RLDR builds this table into each program and the system uses it to store runtime information on the program. START is WRITE's START address. TMIN, the task scheduler, starts at 647_8. The .SAC entries are LDA instructions, not locations. RLDR takes them from the module NSAC3, described above.

Running WRITE

WRITE should run properly now; you've spend enough time on it. Try it out:

```
WRITE)
?
```

You've fixed it; the intial prompt is correct.

```
WRITE RUNS RIGHT, Q.E.D.)
WRITE RUNS RIGHT, Q.E.D.

?

&)
R
```

TYPE W), and find that it contains all messages. WRITE is right, and all ventures into Superedit, the assembler, RLDR and the debugger have paid off. WRITE is a neat little program, and you can use it, or parts of it, to produce impromtu logfiles and do other things.

In the future, if you plan to do a lot of assembly language programming, you might want to create a directory (e.g., ASM.DR) for your assembly language programs, and link to the needed utilities from it. You'll need links to the following files: (N)SPEED.SV/SPEED.ER, ASM.SV/XREF.SV *or* MAC.SV/MACXR.SV/MAC.PS, RLDR.SV/RLDR.OL, and SYS.LB.

End of Chapter

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 CLI commands (e.g., BUILD), Superedit commands (e.g., C), FORTRAN, BASIC or assembly language symbols (e.g., FORMAT, LIST, JSR), pseudo-ops (e.g., .BLK), and system calls (e.g., .CRAND).