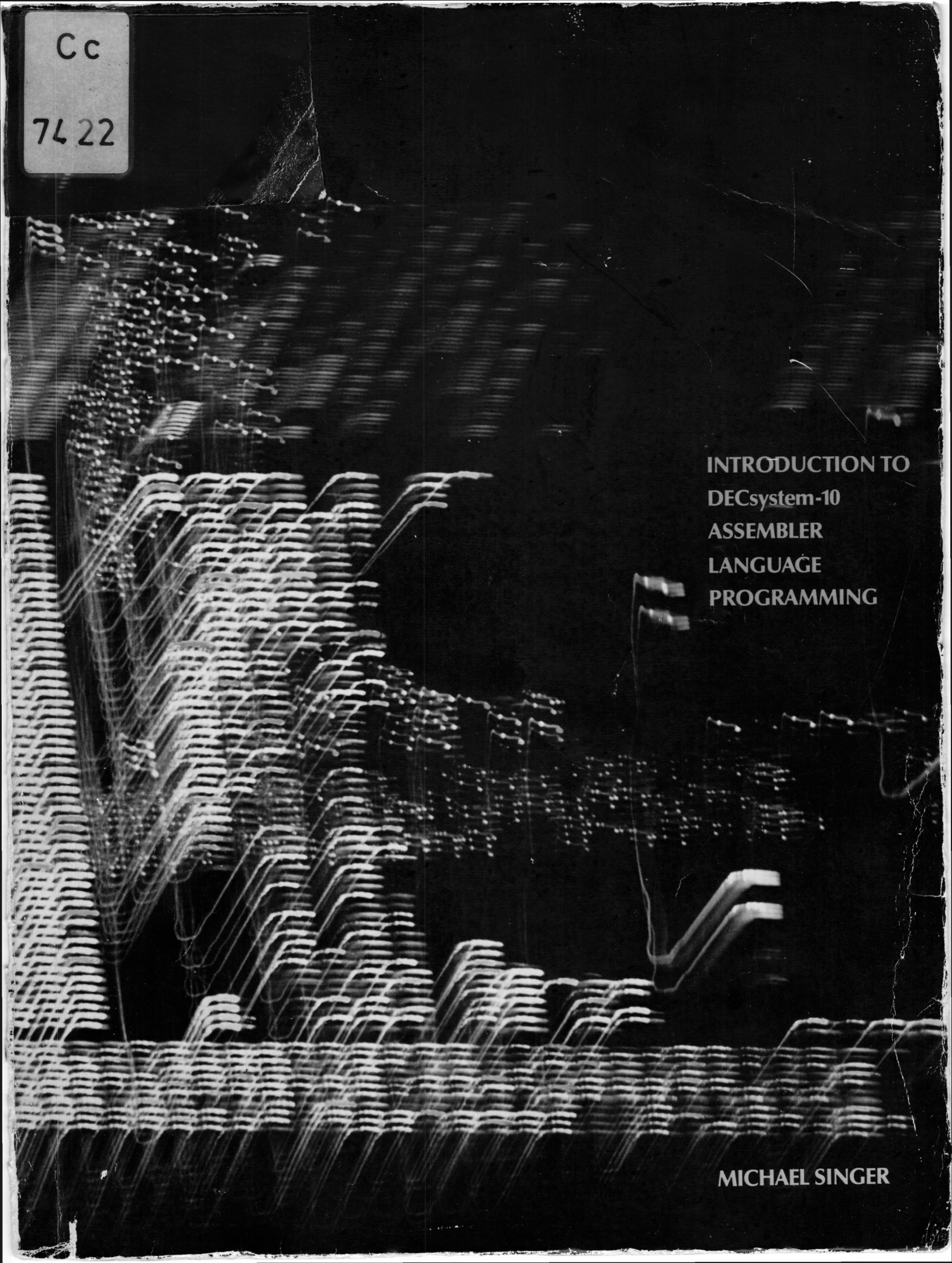


INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

MICHAEL SINGER


DOPPEL

INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

MICHAEL SINGER
Stanford University

JOHN WILEY & SONS
New York/Santa Barbara/Chichester/Brisbane/Toronto

Library of Congress Cataloging in Publication Data:

Singer, Michael, 1942–
Introduction to DECsystem-10 assembler language
programming.

Includes indexes.
1. DECsystem-10 (Computer)—Programming.
2. MACRO-10 (Computer program language)
3. Assembler language (Computer program language)
I. Title.
QA76.8.D4S55 001.6'42 78-8586
ISBN 0-471-03458-4

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

PREFACE

With the widespread availability of higher level languages (such as ALGOL, COBOL, FORTRAN, PL1) for computer programming, as well as packages put out by the major computer manufacturers that, almost at the touch of a button, will perform a variety of complex tasks, it is reasonable to ask why any other than a relatively small number of specialists should trouble to learn assembler language programming at all.

There are good practical, theoretical, and aesthetic reasons for doing so. On some of the excellent smaller machines now being used in scientific and commercial applications, the compiler required to translate a higher level language into machine language would take up so much computer memory space that little would remain for the user. Even when a higher level language is in use, the diagnostic records put out by the machine are typically at the assembler language level. In our opinion, however, the most useful function served by a knowledge of assembler language programming is to give the user a much closer awareness of how the computer works, as well as inestimably greater control over its workings, than is feasible with a higher level language. In our experience, the higher level language user who is familiar with assembler language is a more efficient—even a happier—programmer than the one who is not.

Every computer facility supplies booklets explaining the LOGIN procedure, by which the user gains access to the computer. The novice is then too often left facing across a chasm, beyond which, hopelessly out of reach, lie the manufacturers' manuals and many superb texts on the theory and practice of programming. This book is intended to serve as a bridge across that chasm. It is suitable for use by the higher level language user who would like to learn assembler language; but also, we would like to stress, by the complete beginner with no knowledge whatsoever of computers. The notion that assembler language programming is esoteric and inherently difficult is, in our experience, very much mistaken. On the contrary, for many people it seems to be the natural way to start off with computers.

This book is equally suitable for commercial, scientific, and any other users. The path to an easy-going facility with the basics of the subject is the same for all. There is no shortage of texts and courses dealing with applications to any subject or task the reader may have in mind. But in the first instance, every user must know how to perform input and output, store and retrieve information, and manipulate texts and numbers at an elementary level; for these are the fundamentals of communication with the machine.

All computers have a great deal in common, and much of what is said here applies equally well, with only minor changes, to many other machines. Computer programming is, however, a practical art, and must be learned by continual practice. Because the beginner at the computer terminal is a good deal more aware of the practical differences between different machines than of their structural similarities, we feel that an introduction of this kind should deal specifically with a particular computer system.

Our choice of the DECsystem-10, based on the PDP-10 computer and manufactured by the Digital Equipment Corporation, is no adverse reflection on other machines made by that company or any other company. We do, however, feel that it is a suitable computer on which to base these notes, for several reasons. It is widely and increasingly available, in universities as well as scientific and commercial installations in the United States, Europe, and elsewhere. Its assembler language is very flexible, and is equipped with an excellent utility for tracing and resolving program errors *(debugging)*. Furthermore, it was designed for use *on line;* that is, the user sits at a terminal and converses with the machine, rather than wait patiently while laboriously produced punched cards are processed. And while many machines may be used on line, the design of this one frees the user from the tedious concern with minor details of formatting, such as spacing, needed with machines designed primarily to process punched cards. The assembler language of the DECsystem-10 is commonly known as MACRO-10.

Our approach has the reader writing complete programs, although naturally rather trivial ones, from the very beginning. Thus, access to a DECsystem-10 installation is helpful from the outset. There are no other prerequisites. In our numerous examples we have striven for a combination of comprehensibility and efficiency; but when necessary we have sacrificed the latter to protect the former, for this is a study guide rather than a manual. We request the tolerance of those professionals who cannot abide seeing twenty steps being taken when nineteen would suffice.

Chapter 1 is written with the novice particularly in mind, and the reader with any experience of computers will pass through it rapidly. However,

study with care any statement centered like this one, as it may well be crucial.

Octal and binary numbers must be introduced, and indeed a programmer should ideally be able to think with numbers in any base. Such a facility, however, may be acquired gradually, and so in Section 1.3 we go no further than is necessary to understand what follows. At no stage do we encourage the reader to gain skill in performing calculations in various bases, or in base conversion; in our experience, once the principles are understood, the student's time is better spent in learning how to pass such drudgery to the computer.

Especially in the early stages, the reader may have a sense of being instructed to do things whose function is not fully explained. It is hard to see how this could be avoided. Even the most trivial program requires the support of a very complex system to create and to run it. The beginner must learn the commands that invoke this system in order eventually to gain the experience necessary for a proper understanding of those very commands. We have tried to foster in the reader an approach in which thoughtful endeavor to understand what is presented is balanced by trust that dimly perceived concepts will in due course be clarified.

MACRO-10 is too rich a language to be covered in its entirety in a book of this size. Nevertheless, we have included virtually all the assembler language instructions with full descriptions and many programming examples. The main features of creating macros are covered; so also are FORTRAN subroutines called by MACRO-10 programs, and MACRO-10 subroutines called by FORTRAN programs. The most frequently used monitor calls are discussed, including those handling input/output, terminal control, and enabling traps. This is certainly enough for all normal user programming needs. Those readers who want to proceed further, particularly into systems programming, will be ready after reading this book to refer to the manuals. A warning should be given that much less care goes into preparation of the descriptive literature than into the machine itself and its software, and the manuals contain many obscurities and errors.

There are two appendices. In Section 1.2 we introduce the basic features of the editor TECO. These are sufficient for the needs of this book, and a treatment of some of the more advanced features is relegated to Appendix B. Nevertheless, the reader who studies this additional material will not regret the time spent in acquiring greatly enhanced editing power. Although TECO is the most complex of the DECsystem-10 editors, we feel that it alone is sufficiently comprehensive for the assembler language programmer, whom we would discourage from using any other.

Appendix A treats DDT, the debugging facility of the DECsystem-10. Before the advent of

DDT and similar systems, half of a program could consist of routines to print out information as a check on the functioning of the part doing the useful work. After all the bugs were removed, these routines would be discarded. So the time saved by DDT can hardly be exaggerated. But DDT occupies a more central role in this book; it is a basic tool in our investigation of the workings of assembler language. Consequently, Appendix A is designed to be read in parallel with the main body of the book. A start should be made on it when studying Chapter 2, and a first reading of it is best concluded before beginning work on Chapter 4.

We have endeavored to minimize the possibility of errors, especially in our programming examples. Every complete program in this book has been directly reproduced from computer printout. These programs have all been run, and where relevant tested with a variety of input data. Even our shortest illustrative routines are sections removed from thoroughly tested complete programs. In this way we hope to have spared students one of the greatest frustrations all too often engendered by programming texts.

This book will find its main use as a course text; however, a preliminary version has also been used successfully by individuals working alone. Such persons are strongly encouraged to obtain access to a DECsystem-10; computer time is a readily available commodity, and with reasonable care the cost should be at most comparable with that of class instruction. For all users, it is worth remembering that one of the easiest ways of wasting computer resources is to start thinking out a program after sitting down at the terminal.

The text contains collections of exercises, at least some of which should always be done before reading on. Most of the exercises are straightforward tests of understanding, although the time they require varies greatly. The symbol * marks a few problems of somewhat greater difficulty.

It is a pleasure to acknowledge the encouraging comments and suggestions of students and colleagues, past and present. Dr. David Ford of Ohio State University was especially helpful during the early stages of manuscript preparation. Thanks are owed also to the University of Pennsylvania for its generous provision of facilities, and to the staff of John Wiley & Sons for their understanding support.

MICHAEL SINGER

CONTENTS

INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

CHAPTER ONE

PRELIMINARIES

1.1 THE TERMINAL

The computer terminal is rather like a typewriter, but with a few special features. There are a number of different types of terminal; some display characters typed by the user, and the output of the computer, on a TV-style screen rather than on the more usual paper roll. There are certain special characters located in different places on different terminals, so the reader should spend a few minutes in becoming familiar with the characteristics of any new or unfamiliar model.

As on a regular typewriter, there is a SHIFT key. It should be observed, however, that many terminals have only uppercase (capital) letters available. This need not trouble the programmer since computer instructions do not distinguish between upper and lowercase letters. With these terminals, do not use the SHIFT key to obtain regular letters, because other characters will sometimes result. For example, on some terminals @ is SHIFT-P,] is SHIFT-M, while [is SHIFT-K. Anything of this kind is normally clearly marked on the respective keys.

Be careful always to distinguish: I (capital i), l (lowercase L) and 1 (one); O (capital o) and 0 (zero); parentheses (. .) and square brackets [. .].

An important feature is the CONTROL key. Like SHIFT, this does nothing on its own; but when held down while other keys are struck, it produces a whole new set of characters. Some CONTROL characters are just plain characters. If you type CONTROL-A, you will just see ^A appear on the paper or screen. If, however, you type CONTROL-C while a program is running, ^C will appear, and in addition the program will stop (if calculation is in progress two ^C are needed for this effect). Several other CONTROL characters have "break" or "interrupt" functions, which we shall study individually as we need them.

In this book CONTROL will be denoted by the ^ symbol. *Warning:* this is not the "up-arrow," or on some keyboards "circumflex" symbol (this symbol is often SHIFT-N). Typing ^, followed by C, will also appear as ^C, but will not have the special effects of CONTROL-C.

in this book ^A etc. always means type the character while holding down CONTROL, unless specifically stated otherwise.

On keyboards without a special tabulator key, ^I produces a tab, normally set at every eighth space.

The ESCAPE (also known as ALTMODE) key has special functions that we explore in the next chapter. Observe carefully that, although striking it produces a $ sign, it is not the same as the key marked $. They produce the same symbol on the paper, but not at all the same internal effects. The same danger of confusion exists as with CONTROL and up-arrow. In this book

$ always means ESCAPE key, unless specifically stated otherwise.

If you type a wrong character by mistake, press the RUBOUT (also known as DELETE) key. You will see the wrong character appear once more; this may appear preceded by a \ sign. This indicates that the character will not be transmitted to the computer. You can *not* delete characters by backspacing and "typing over." Any number of characters can be deleted by pressing the RUBOUT key the appropriate number of times. Remember that spaces are characters too!

It may be easier to delete a whole line and start again. Typing ^U (remember that this means CONTROL-U) will delete the line you are currently typing. The machine will automatically move on to the beginning of the next line on the paper.

To start your session, type ^C. This ensures that you are in communication with the *monitor.* The monitor may be regarded as the organizing center of the computer. You know that you are dealing with the monitor when your terminal of its own accord goes on to the beginning of the next line, and types a period

You now type LOGIN, using the RUBOUT key to rectify any errors. But nothing will happen until you press the CARRIAGE RETURN key, denoted here by ↵, for only then is the whole line that you have typed sent to the monitor. The response is

#

whereupon you type in the identifying number issued to you, followed by a ↵. Then

PASSWORD?

is self explanatory. Note that what you now type is not *echoed,* to preserve secrecy of your password. Any messages from the (human) operator to all users will now appear, after which you will see

which indicates that the monitor is ready for your instructions.

You have now started a *job.* As part of a job you may write and run any number of programs. The job goes on until you *kill* it. This must be done by giving the monitor a specific instruction. It is not enough just to switch off the terminal and walk away. On some installations a job is killed automatically if there is no activity for some time; but on others a job continues, and accumulates connection charges indefinitely.

Although we have done nothing constructive yet, it is as well to learn immediately how to kill a job. The first thing to do is to get in touch with the monitor. If a period has just been typed by the machine at the beginning of a line, the monitor is already waiting for an instruction. Otherwise, typing ^C twice will always cause the monitor to intervene and stop whatever else is going on in your job, and type a period. Now you type KJ/F followed by ↵ to kill the job. KJ is a mnemonic for Kill the Job. Various letters can follow after /, but an F ensures that nothing you may have put into store is destroyed. A message will appear detailing how much time you have used. In some installations, you will also be told how much money you have spent.

Exercise: Practice starting and killing a job using the RUBOUT (or DELETE) key and ^U, and using ^C.

You do not have to LOGIN for the remainder of this section.

Another useful CONTROL key is ^O. If you are not interested in whatever is being typed out at your terminal, ^O will stop the output. Try giving to the monitor the command SYSTAT followed by a ↵. The monitor will type out information about the current usage of the system; when you have seen enough, type ^O.

Make sure the SHIFT key is not down, and type a letter of the alphabet. If an uppercase letter appears, get in touch with the monitor and give it the instruction SET TTY LC followed by a ↵. TTY is a standard code representing the terminal, and LC is the mnemonic for lowercase. There must be at least one space between SET and TTY, and between TTY and LC. You will now be able to type lowercase letters, as long as your terminal is equipped to produce them. If you later give the monitor the command SET TTY UC only uppercase letters will then be available. Observe that these commands have no effect on the action of the SHIFT key to produce symbols other than letters of the alphabet.

Press the TAB key. If nothing happens, you must tell the monitor that your terminal does not itself produce tabs, by entering SET TTY NO TAB followed by ↵. This command looks paradoxical, but there is logic in it nevertheless.

Try SET TTY NO ECHO ↵. To undo the effect of this, issue the monitor command SET TTY ECHO ↵. Since this time you cannot see what you are typing in, before entering the line with ↵ type ^R. This character will always have displayed for you the line you are currently typing to the monitor. Correct any errors with RUBOUT and enter the line with ↵.

Now LOGIN, and go on to the next section.

1.2 THE EDITOR

The function of the editor is to render what you type at the terminal into a form with which the machine is equipped to deal. In other words, you use the editor to create a *file.* Some of your files will be lists of instructions to the computer—that is, *programs.* Others may be collections of data to be processed by programs.

The editor will also transfer your file from the temporary storage area (*memory*, or *core*) in which it is housed as you write it, to permanent *disk* storage.

Since we do not yet know how to issue instructions to the computer we cannot write a program; we can nevertheless write a simple file.

Let us write a file called, for example, TEST, which will contain the information.

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

To summon the editor to make a new file, we type after the period issued by the monitor the command MAKE TEST , followed by a carriage return. Remember that the initial period comes from the monitor, not from you. We shall stress that something is typed by the machine rather than by the user by underlining it. The underlining does not appear at the terminal. So what happens is

.MAKE TEST ↵

↵ indicates that you press a CARRIAGE RETURN. Do not type a period after TEST ; there must be at least one space between MAKE and your program name, but more will do no harm. Your program name can be any collection of up to six letters and numbers that you care to choose, as long as the first character is a letter.

The machine will now print an asterisk

*

Output of an asterisk tells you that you are in contact with TECO, the editor. TECO understands a wide range of commands, enabling you to insert, amend, or delete text with great ease.

Warning: TECO commands are letters of the alphabet, and it is very easy to confuse them with the text of the file. TECO command strings in this section should be studied with the greatest care,

letter by letter. Your own command strings should be typed with similar care, *and re-read before being entered* (with $$ as explained below). Be careful: a wrongly typed letter might be a command you do not yet know that could destroy your entire file!

What you have created so far is an empty file named TEST. To insert text, use the TECO command I followed by the text you want to write. Finish your text by pressing the $ (ESCAPE) key. Everything between I and $, including spaces and carriage returns, becomes part of your text. So the line looks like this

```
*ITHE QUICK BROWN FOX JUMPS OVER THE LAZY DOG$
```

If you make a mistake while typing, use the RUBOUT (or DELETE) key to erase individual characters. To delete the line on which you are working, use ^U. The terminal will go on to the beginning of the next line, and you will get a new asterisk. Your session might go something like this

```
*ITHE QYICK VRO^U
*ITHE QUICK BROVVWN FOX JUMPS OVER THWWE LAZY DOG$
```

You become disconcerted by all the mistakes on the first line, and use ^U so that you can start again. Remember that this erases the whole line, which includes the I command; so you need to issue another I command before your text. In the next line you accidentally type V in place of W, and W in place of E. Both of these are corrected using the RUBOUT key, which echoes the original error.

This is all you planned to put in your file, so you can exit from TECO. The command EX does this. To actually get your commands performed, however, you must type $$ (ESCAPE twice in succession). This instructs TECO to carry out all the commands you have issued (since the last $$, if any). So your whole session, if no errors were made, would look like

```
*ITHE QUICK BROWN FOX JUMPS OVER THE LAZY DOG$EX$$
.
```

As you see, EX$$ takes you back to the monitor.

If you forgot the $ before EX, the final $$ would cause the performance of the instruction to insert the text

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOGEX
```

and you would still be with TECO.

Back with the monitor, type DIR to list the *directory* of your files.

```
.DIR↵
```

As you see, TEST is there! To find out what is in TEST

```
.TYPE TEST↵
```

and see for yourself.

Suppose you want to amend what you have written in your file. If you have exited from TECO, you get back like this

```
.TECO TEST↵
```

Perhaps you want to change JUMPS to JUMPED. This is done by the FS command. You follow FS by old text, then $, then new text, then $ again. So you could type

```
*FSJUMPS$JUMPED$
```

or, more economically,

```
*FSPS$PED$
```

or even

```
*FSS$ED$
```

It will always be the first occurrence of the text that is changed, starting from wherever the editor's file position indicator, or *pointer* is set. Calling in TECO for an already existing file sets the pointer to the beginning of the file.

After changing JUMPS to JUMPED, the pointer is set after the final D of JUMPED. The command T will type out a line from the pointer to the end of the line, but

```
*FSJUMPS$JUMPED$T$$
```

results in the editor typing out

```
 OVER THE LAZY DOG
```

To see that you have in fact made the proper correction, set the pointer to the beginning of the line with the command 0L (remember that 0 is zero, not letter O). So the whole command string is

```
*FSJUMPS$JUMPED$0LT$$
```

Notice that the concluding $$ is necessary to actually get things done! It is the command to carry out the instructions that until this point have merely been noted.

Perhaps you would like a period after DOG? Use the S command to search for G (there is only one G; but if there were more, you could always search for OG). This sets the pointer after G, so you can insert your period. Notice that with S , you end the text with $, just as with FS and I.

```
*SG$I.$0LT$$
```

will have the line typed out as you want it.

Perhaps you dislike the format? A new line after JUMPED might be more pleasing. No problem.

```
*SED$I
$0LT$$
```

After I the required text was just a ↵, which is exactly what gets typed by you at the terminal. The editor's response is now

```
 OVER THE LAZY DOG.
```

because T types the current line; and after inserting the ↵ the current line is now the second line of our text. Notice that we forgot to delete the space between JUMPED and OVER. Since the text is already entered in the file, the RUBOUT key no longer works, as the function of RUBOUT is to prevent the character just typed from being entered. However, the D command deletes the next character after the position of the pointer. So in place of the previous command sequence

```
*SED$I
$D$$
```

would give us the text we want, and the pointer is set to the beginning of line two of our file. To check, set the pointer back a line with the command −L, and type two lines with the command 2T. Observe that

the T *command types from the position of the pointer, but does not itself move the pointer.*

After carrying out the last command,

```
*−L2T$$
```

yields output of

```
THE QUICK BROWN FOX JUMPED
OVER THE LAZY DOG.
```

The pointer is once again at the beginning of the file. Observe carefully that the same type-out is obtained, starting with the pointer at the beginning of line 2, by

```
*-TT$$
```

but this does not move the pointer at all.

With the pointer at the beginning of line 2, the command string

```
*FSQUICK$QUICKEST$$
```

would produce something like this:

```
?SRH     Cannot Find "QUICK"
```

because the editor *searches only from the pointer onwards.* (Note that an unsuccessful S command sets the pointer back to the beginning of the file.) So first set the pointer back a line by −L; or, to save the trouble of counting, simply set the pointer to the beginning of the file, using the command J

```
*JFSQUICK$QUICKEST$0L2T$$
```

produces what you wanted, and types out both lines.

Suppose you now change your mind about spreading the text over two lines. So let us delete the ↵↓. No problem, as long as we are aware that

pressing the CARRIAGE RETURN *key produces two characters; first a carriage return, then a line feed.*

Carriage return is the mechanism that sends the terminal back to the start of the same line; line feed moves it down one line.

The correct command string is thus

```
*SJUMPED$2DI $$
```

in which we remembered to replace the space. And now

```
*0LT$$
```

will type out the whole text, once again on one line.

In the unlikely event that it is needed, a carriage return alone is produced by typing ^M. This returns the terminal to the beginning of the current line. To advance a line, the LINE FEED key, or equivalently ^J, will serve. Normally, of course, the carriage return key is used — but remember, for editing purposes, that it "echoes" a line feed.

Exercise: Practice using these commands with various texts.

Any whole number, positive or negative, may be used with D, L, and T. Note that L and T are for lines, while D is for characters. To delete whole lines use K, with or without a whole positive or negative number preceding it.

Observe that −25L sets the pointer 25 lines back. If there are not that many lines, the pointer is set to the beginning of the file. 30L advances the pointer 30 lines, or, if there are not that many lines left, to the end of the file.

Counting back from the end is easier if you end your file with ↵↓.

So, in our example, we would insert text as follows

```
*ITHE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
$
```

This method is preferable. Using it, you can have the last line of a file (of less than 100 lines) typed

out by

```
*100L-T$$
```

Of course, 100 can be amended as necessary. The pointer would now be in the correct position to append new lines of text to the file.

If your file is long, TECO will load only the first part of it unless instructed otherwise. So your first command to TECO should be A, which causes more of the file to be loaded at once. If more core space is taken up in doing this, you will be informed accordingly.

```
*A$$
[3K Core]
```

When using D, remember that spaces and punctuation marks are also characters, and that ↵ comprises two characters. A TAB, or ^I, is one character. −3D would delete the three characters preceding the position of the pointer; 100D deletes the 100 characters following the position of the pointer. −3D100D or 100D−3D would do both.

With K it is most important to be aware of the position of the pointer. K will delete the current line from the position of the pointer to the end, including the ↵ that terminates the line. To delete the whole line use 0LK. 3K will delete from the pointer to the end of the current line plus the whole of the next two lines. −K will delete the whole of the previous line, −2K the previous two lines, and so on; in addition, the current line will be deleted, up as far as the pointer.

T will type out whatever K would delete.

Suppose that throughout your file you have written THRU, and would now prefer to have THROUGH. Estimate beyond the maximum number of times THRU occurs, say, 1000 times. Put the command you want between brackets like this <...>, with 1000 in front of it and $$ after it, and the command will be carried out as many times as possible up to 1000. (You will not be informed how many times it was actually possible to carry it out.)

```
*1000<FSTHRU$THROUGH$>$$
```

However, since this will result in changing any occurrence of THRUSH to THROUGHSH, more care is needed.

Exercise: Devise a foolproof way of doing this. (Observe that the separate word THRU can be followed by only a few possible characters, such as space, comma, period, ↵, and so on.)

To delete a given text string without troubling to set the pointer, use FS to replace it by a *null text*. For example, if the first occurrence in your file of IT IS A FACT THAT is superfluous (including the space after THAT), then

```
*JFSIT IS A FACT THAT $$
```

will delete it. Since the form of this command includes two successive $ characters (to *delimit* the null text), this command is carried out at once and you get the response of a new line and a fresh asterisk.

A final words about the RUBOUT key. Suppose you type

```
*ITHE QUICJ
BROW
```

and notice your error only now. You can simply carry on, and later use

```
*JFSCJ$CK$$
```

or, since you have not yet had the current command performed by issuing a $$, you can RUBOUT all the way back to J, then re-type. As you press the RUBOUT key, the characters deleted will be echoed. This just looks a little strange at first with spaces, TAB, and ↵.

In our example, ^U would merely delete BROW. If you prefer to delete the whole of the

current command (back to after the last $$, if any), type ^G twice; this character also rings the margin warning bell, which you will hear as you type it! You will be issued a new asterisk, and can start your command again.

After all this amending, if you ask the monitor to list your directory, you will find not only TEST but also TEST.BAK, the *back-up* file which is created while you are amending an already existing file. Have it typed out by

```
.TYPE TEST.BAK↵
```

and see for yourself. You must enter TEST.BAK exactly like that, with the period, and with no spaces around the period. The file name TEST has now acquired the *extension* .BAK . This is one of many file name extensions that convey information, to both user and machine, regarding the file.

If you have no further use for a file such as, say, TEST.BAK, you should

```
.DELETE TEST.BAK↵
```

You type the word DELETE, rather than press the RUBOUT (or DELETE) key. You should always conserve disk space by deleting superfluous files.

Exercises: (i) Create a file containing the text

```
ALL THAT GLISTERS IS NOT GOLD
                        SHAKESPEARE
                              1596
```

and exit from TECO.

(ii) Amend the file to contain

```
ALL IS NOT GOLD THAT GLISTERS
                        CERVANTES
                              1615
```

and exit from TECO.

(iii) Amend the file again to contain

```
ALL IS NOT GOLD THAT GLISTENETH
                        MIDDLETON
                              1617
```

and exit from TECO.

*(iv) If your terminal has lowercase letters available, change all but the first letters of each word in the file to lowercase. (In a search command, the text is searched without regard to upper or lowercase.)

(v) Devise a single sequence of TECO commands to change a file
(a) from single line spacing to double line spacing;
(b) from double line spacing to single line spacing.

(vi) The C command moves the pointer forward one character. It may be preceded by a positive or negative number, to specify how far, forward or backward, the pointer is to be moved. Write a file in a "secret" code, as follows: replace all spaces by letter S, all ↵ by letter C; insert a ↵ after every fifth character; type out the lines of the file, starting from the last line and working backwards (using one command string); retype the file in this form, and destroy the old version.

Can you now "decode" the file? If not, improve the coding method so that you can.

(vii) Reduce the storage space taken up by a file, by allowing only one space between words, after punctuation marks, and at the start of a line to indicate a paragraph. Also, remove any blank lines.

(viii) Restore the format of a file treated as in the last example. Allow two spaces after a semicolon or colon, three after a period. Indent new paragraphs five spaces, with a blank line preceding.

1.3 OCTAL NOTATION

A computer is a machine that deals exclusively with numbers. In order to instruct a computer to carry out an operation, the operation itself must be encoded as a number meaningful to the computer. Letters of the alphabet, as part of the text of a file, must also, somehow, be encoded as numbers. Much of the encoding process is done by the machine without necessitating the programmer's concern; but we do need to consider not only how the computer encodes alphabetic and other symbols as numbers, but also the way in which it registers numbers themselves.

A computer does not have ten fingers. As a result, the number nineteen, say, is not considered by the computer as being in any essential way one ten plus nine ones. The computer does not "think decimal." In fact, the computer "thinks *binary,*" that is, in the number system in which two replaces ten as *base.* In such a system, instead of successive columns, from right to left, denoting units, tens, hundreds, thousands, and so on, they represent instead units, twos, fours, eights, sixteens, and so on.

In binary notation, since nineteen is equal to sixteen plus two plus one, it is represented by 10 011. Just as with decimal representation, we group digits in threes for ease of reading. We write this succinctly as

D 19 = B 10 011

where D stands for decimal, B for binary. In the binary representation, observe the 1 on the left in the sixteens column, 0 in both the eights column and the fours column, 1 in the twos column, and 1 in the units column.

These are merely two different ways of representing the same number; one is more convenient to a human being with ten fingers, the other more convenient to a machine with electrical switches, or other devices, that have just two "states" (for a switch, the two states are ON and OFF).

The trouble with binary notation is that even quite small numbers are very unwieldy for human beings to read and interpret. For example, not only is it tedious to find the decimal equivalents of 10 010 110 101 and 10 010 101 101, it takes more than a glance to see even that they are distinct numbers! The decimal equivalents are the much more compact 1205 and 1197.

Exercise: Have a try at checking out the equivalence between these binary and decimal representations.

Nevertheless, communication between user and machine must take into account that the machine holds numerical information in binary notation. The machine with which we are dealing has as its number holding unit the *word,* each of which contains thirty-six individual binary digits; that is, thirty-six positions each of which can represent a 0 or a 1. "Binary digit" is abbreviated to *bit.* You can see from our discussion above that eleven bits are needed to represent D 1205, five for D 19.

There is a special code, which we shall learn later, that instructs the machine to interpret the following number as a decimal number. Since performing tedious calculations is the job of a machine rather than of a human being, we would, for example, write in 1205 as a decimal number, instead of laboriously converting it into another base.

We do, however, need to know more about how the machine holds information within its words, in order to take full advantage of the power of assembler language. To make this somewhat easier, the machine is set up to deal readily with numbers not only in binary form, but also in *octal* form, in which the base is *eight.*

It is very easy to convert binary representation to octal. Consider again B 10 011. Notice that B 011 is D 3, which is the same thing as octal O 3 (counting to three is the same process with eight fingers as with ten). B 10 is D 2, so also O 2; but because there are three more columns of binary digits remaining to the right, this actually means O 2 multiplied by 2 × 2 × 2, that is, by eight. So B 10 011 = O 23, because in base eight, the digit 2 is in the position that means "multiply by eight."

We worked out earlier that this is D 19, which is also obvious from the octal notation: twice eight plus three equals nineteen.

Consider again D 1205 = B 10 010 110 101. The binary triads are, from left to right, O 2, O 2, O 6, O 5. So the octal representation of this number is O 2265, which means 2 × (eight × eight × eight) + 2 × (eight × eight) + 6 × (eight) + 5.

To convert octal to binary, reverse the process. For example, O 734: O 7 = B 111, O 3 = B 011, O 4 = B 100. So O 734 = B 111 011 100.
In decimal notation, this is 7 × (8 × 8) + 3 × (8) + 4 = D 476.

It is important to be alert to:

D 10 = O 12
D 8 = O 10
D 64 = O 100.

Octal representation is the normal mode in which the machine regards a number. Anything else must be specifically declared.

Exercises:

(i) Why is it so easy to convert between base two and base eight?
(ii) Is it equally easy to convert between
  (a) base three and base twelve?
  (b) base three and base nine?
  (c) base two and base six?
  (d) base two and base four?
  In the cases where conversion is easy, describe how it is done.
(iii) If you were asked to convert the base seven number 59 to base ten, what comment would you make?
(iv) Convert to decimal representation
  (a) O 37; (b) O 40; (c) B 1 111; (d) B 11 110.
(v) Convert to octal representation
  (a) D 37; (b) D 40; (c) B 1 111; (d) B 11 110.
(vi) Convert to binary representation
  (a) D 37; (b) O 37; (c) D −32; (d) O −32.
(vii) What is O 100 − 1
  (a) in octal notation?
  (b) in decimal notation?
(viii) Is O 15 − O 60 positive or negative? Why?

Now we are ready to discuss how the machine encodes the various symbols that appear on the keyboard of the terminal. There is a comprehensive code in which to every single symbol there corresponds a number. This code, which is widely used on many different machines, is called the American Standard Code for Information Interchange. It is commonly referred to by its acronym ASCII (pronounced az-key). On the full standard terminal there are in all 127 distinct symbols (this is D 127). This includes not only upper and lowercase letters, numerals, and special symbols, but also special combinations such as CONTROL characters, which are regarded as one symbol by ASCII. When you press a key on the terminal, the corresponding ASCII code number is electronically transmitted to the monitor. For example, the ASCII code for ^A is 1. Suppose that we have somehow contrived to make a certain word in the computer contain the number 1. By this we mean that, reading from right to left, the first bit in the word is set to 1, and all the rest are 0. Depending on what we are doing, we might want this 1 to mean ^A, or we might want it to mean simply the number 1. It is important to realize at the outset that the computer cannot "know" which we mean until we instruct it accordingly.

Let us now write our first, very simple program. We shall instruct the machine to print the letter B , then stop. We need to know the ASCII code for B, which is 102. The program is very short, but contains several new things, which we shall examine one by one.

```
START:  OUTCHR  [102]
        EXIT
        END     START
```

The very first word of our program, START, is, in spite of appearances, not an instruction to the computer. It is merely a *label,* which serves only to identify the line on which it is found. When a line has a label, the label must be on the extreme left and followed by a colon, with no intervening space. After the colon there must be at least one space, but it is convenient to reserve a column for labels as we have done above, using the TAB key freely to obtain an easily readable format.

We label this line because we want to refer to it later in the program. To see where we refer to it, look at the last line of our program. END , which is assembler language terminology, is the indication to the machine that no further instructions or designations are to follow. What follows END , on the same line, is a direction to the computer as to where operations are to commence when the program is executed. In our program, this is to be at the line labeled START .

The choice of label is virtually at our disposal. We may use any combination of up to six letters and digits, as long as the first character is a letter. START is an obvious and suggestive candidate.

OUTCHR is an instruction to the monitor to send the contents of the appropriate word to the terminal, as an ASCII character. What is the appropriate word? That is what the square brackets [...] are for. The assembler will find a word within the computer, and set its bits to represent what it finds between the brackets; in other words, it will create an *address* for the data between the brackets.

EXIT is a necessary part of the program. It instructs the monitor to perform certain routine functions necessary to stop the program, and then to stop it. If you write a program that reaches its END statement without first encountering EXIT , you will get an error message when you try to execute the program. (Try it!)

You can see below a reproduction of the terminal session in which the above program was created and executed. We called the program TEST.MAC . TEST was chosen as a name for obvious reasons. The extension .MAC should always follow the name of any MACRO (assembler language) program, as it enables us to use a very simple procedure to execute the program. Nothing should come between the program name and .MAC , exactly as shown.

To remind you that it is up to us to choose labels, we used a different one for the line at which operations are to commence.

The command to the monitor to execute a program is EXECUTE, which is conveniently abbreviated to EX ; this should not be confused with the TECO exit command. Notice that it is not necessary to use the file name extension in the EX instruction.

```
.MAKE TEST.MAC

*ICOMNCE: OUTCHR  [102]
          EXIT
          END     COMNCE
$EX$$

.EX TEST
MACRO:  .MAIN
LINK:   Loading
[LNKXCT TEST Execution]
B
EXIT
```

We now know that the ASCII code for B is 102. We also need to be aware that this means *octal* 102. The ASCII code interprets the symbols of the terminal as octal numbers between 0 and O 177.

Observe that D 128 is 2 × (eight × eight), and so is equal to O 200. Subtracting 1 from this number gives D 127 = O 177 as the number of distinct ASCII symbols.

It is important to understand why O 200 − 1 = O 177.

Consider what happens when we try to add 1 to O 7. The quantity in the units column is increased to eight, so we must carry to the left, into the eights column. So O 7 + 1 = O 10. Similarly, O 17 + 1 = O 20. If we try to add 1 to O 77, carrying 1 into the eights column increases the quantity there to eight, so we must carry one more column to the left, into the (eight × eight) column. So O 77 + 1 = O 100. Similarly, O 177 + 1 = O 200, and so on.

Of course it is possible to convert these ASCII codes into decimal representations, but this is not necessary. It is a much better idea to get used to these numbers in the octal form in which they are always quoted. Just remember that no digit may exceed 7; and so adding 1 into a column with a 7 in it produces 0, with a 1 carried to the left.

The ASCII codes for A through G are

A	101
B	102
C	103
D	104
E	105
F	106
G	107

What do you suppose is the ASCII code for H? As you have doubtless guessed, it is the code for G increased by 1; which of course means

H	110

and so on sequentially through to

W	127
X	130
Y	131
Z	132

The numerals on the terminal have as ASCII codes

0	60
1	61

and so on, through to

7	67
8	70
9	71

It is very convenient that the ASCII codes for successive numerals are themselves successive octal numbers; observe that one can be obtained from the other by adding or subtracting O 60.

Let us write a program that will add 1 to a number to be chosen by us at execution time. To instruct the machine to get from the terminal a character that we type in at the appropriate moment, we need the command INCHWL . This instructs the monitor to take in a character, in the "wait on line" mode—after typing in the characters, the machine does not receive them until you enter a ↵. There is another instruction that sends characters as you type them, but it has the disadvantage that if you make a mistake you have no opportunity to amend it with the RUBOUT key.

Our program is

```
START:  INCHWL
        ADDI    1
        OUTCHR
        EXIT
        END     START
```

The command ADDI is used for ADDition in the "Immediate" mode; that is, when the number at the end of the line is the actual (octal) number to be added on. You might wonder what other mode of addition there could possibly be, but the answer to this must wait awhile.

Let us consider what happens when we create this program with TECO (you must choose a name for it), then EXecute it. Nothing at all will happen, after the message

[LNKXCT TEST Execution]

until we type in a character. Suppose we type in 5. Then the ASCII code for 5, which is O 65, is taken in by the machine; 1 is added to it, yielding 66; this is the ASCII code for 6, so the OUTCHR command causes 6 to appear at the terminal, whereupon the machine will EXIT.

Notice that this program does not manipulate the numbers we type in; rather, it manipulates their ASCII codes. It will add 1 perfectly well to numbers 0 through 8. But if we type in 9, it will return the character whose ASCII code is O 72. Try it, and discover which character has that code. If we type in 10, the first character we type is 1; the program will therefore print out 2. The remaining 0 will puzzle the monitor, which will consequently print out, after exiting from the program, the message

.?0?

Even if we could somehow carry out the process of adding 1 successively on each digit of 10, this would not achieve the result of adding 1 to the number D 10 itself. There is no reason why it should: we have given the machine no indication that 10 is the representation of a number in some "positional" notation. Until we do so, 10 is simply entered as symbol 1 followed by symbol 0.

Since the above program manipulates ASCII codes, we can enter any symbol. If we enter A, then B is printed out. B would result in C, and so forth. Entering Z (ASCII code 132) results in [(ASCII code 133).

What follows is an incomplete program fragment, which does nothing at the terminal. It merely reads a numeral between 0 and 9, which we type in at the terminal, as the actual number, not its ASCII code.

```
INCHWL
SUBI        60
```

The second line is SUBtract Immediate: the number O 60 is subtracted from whatever is there already. If we have typed in 7↵, its ASCII code of 67 will have been entered by the INCHWL command. The subtraction command reduces this to 7.

Suppose we type in 8 ↵; then O 70 is entered. The subtraction command reduces this to O 10, which, again, is D 8.

With practice, you will soon find yourself using the SUBI 60 command to convert the ASCII representation of a digit to the number itself virtually automatically.

Of course, to convert back to ASCII code before using OUTCHR, the quantity O 60 must be added back on. This is done by the command ADDI 60 .

The following complete program doubles a number. We introduce the IMULI command, for Integer MULtiply, Immediate.

```
START:  INCHWL
        SUBI    60
        IMULI   2
        ADDI    60
        OUTCHR
        EXIT
        END     START
```

Let us follow through what happens when 3 is typed in. Its ASCII code of O 63 is entered. From this, subtraction of O 60 yields 3. Multiplication by 2 gives 6. Adding O 60 gives O 66. This is the ASCII code for 6, so 6 is printed out. The machine will now EXIT.

This program works perfectly well on 0, 1, 2, 3, 4. What happens if we type in 5? Its ASCII code of O 65 is entered. Subtraction of O 60 gives 5. Doubling 5 gives D 10; remember that this is O 12. Adding O 60 gives O 72. This is the ASCII code for the symbol : which is therefore printed out. It is clear that dealing with numbers that run into more than one digit will require a certain amount of care! Observe that, in the last example, the machine does indeed contain the

correct result of doubling 5. The problem comes when we try to print it out in the usual positional notation by which we represent numbers.

Typing in other symbols with this program yields meaningless, but nevertheless instructive results. Type in A, so that O 101 is entered. Subtracting O 60 from this gives O 21 (why?). Doubling gives O 42. Adding O 60 gives O 122 (why?). This is the ASCII code for R, which gets printed out.

It must by now be getting tedious to have to EXecute the program for every single input. It is also wasteful, and we shall learn how to overcome this later.

Exercises:

(i) Write a program to print out the text PROGRAM.

(ii) Write a program to accept input of a character, then print out THE CHARACTER YOU TYPED WAS followed by that character.

(iii) Write a program to treble a number typed in at the terminal. For what range of input does your program work? What result does your program yield when p is input? Why?

(iv) Write a program to accept input of a two digit number, add one to the number, and print out the result. How do you explain your program's action
   (a) when the second digit is 9?
   (b) when a single digit number is input?

CHAPTER TWO

FUNDAMENTALS

2.1 THE ACCUMULATORS

In the previous chapter we learned the command INCHWL , which instructs the monitor to take a character typed in at the terminal and hold it in the computer. Our programs will be very trivial, however, if we can only hold one character at a time in the computer. In fact, there is room in the computer's *memory,* or *core,* for a program to have at its disposal many thousands of locations, or *addresses,* in which characters may be placed.

Sixteen of the memory locations available to the user are called *accumulators* and are of particular importance. Most of the arithmetical operations can be performed on a number only if that number is held in an accumulator. If we want, for example, to double a number currently held in some memory location other than an accumulator, we must do as follows: move the number to an accumulator; double it in the accumulator; move the new contents of the accumulator back to the memory location.

The sixteen accumulators are numbered, and are identified by their numbers. Note carefully that

accumulators are numbered octally, starting at 0: 0,1,2,3,4,5,6,7,10,11,12,13,14,15,16,17.

If no accumulator number is given in your program, the assembler will assume that accumulator 0 is intended. Thus in Chapter 1, the "action" took place in accumulator 0.

Let us rewrite the program of Section 1.3 that doubles a number between 0 and 4, using accumulator 1 instead of accumulator 0.

```
START:  INCHWL  1
        SUBI    1,60
        IMULI   1,2
        ADDI    1,60
        OUTCHR  1
        EXIT
        END     START
```

Notice how, in the second, third, and fourth lines, the number of the accumulator comes before any other number, and it is followed by a comma if there is more to come on that line.

Be careful not to be confused by a line like

```
IMULI        1,2
```

in which 1 is the accumulator number, and 2 is the actual quantity by which the contents of accumulator 1 are to be multiplied. A good way to avoid confusing the roles of the numbers in lines like this is to give names to the accumulators your program is going to use. Names are of your own choosing: up to six letter or number characters, starting with a letter. It is useful to choose a name having some association with what you are doing. Let us suppose that we want to use accumulator 1 again, and that we will give it the name INT. We must declare INT=1 before the program instructions; then, whenever the assembler encounters INT, it will understand that 1 is meant. The above program now looks like this:

```
        INT=1
START:  INCHWL  INT
        SUBI    INT,60
        IMULI   INT,2
        ADDI    INT,60
        OUTCHR  INT
        EXIT
        END     START
```

The declaration INT= which is then followed by the accumulator number is one of the few places where putting in spaces is not allowed. The = sign must follow the chosen name immediately.

Having the accumulators at our disposal will now enable us to increase the scope of the above program. First, we shall amend it to deal with a two-digit output.

Suppose that our input is 9. Then after the line IMULI INT,2 accumulator INT contains the number eighteen. But ADDI INT,60 followed by OUTCHR INT will lead to print out as follows: D 18 = O 22, O 22 + 60 = O 102, and 102 is the ASCII code for B (try it!).

In order to print out the number eighteen in the form 18, we must examine the meaning of this decimal notation. The symbol 1 gives the number of tens in the number eighteen; the symbol 8 tells us how many units are left over. If we divide eighteen by ten, the result is 1, with remainder 8.

There is a division instruction available that is perfect for our requirements. Suppose we have a number stored in an accumulator, say, in accumulator 2. Then

```
IDIVI        2,5
```

will divide that number by 5, leaving the whole number quotient in accumulator 2; the original number is lost in this process.

But the division instruction does something else at the same time, which is very useful for our purposes: the remainder in the division calculation is put in the next accumulator. In our example, this would be accumulator 3.

Don't forget that carrying out a division on the contents of accumulator 7 puts the remainder in accumulator 10 (the next one!). Accumulator 17 has no next one; if its contents are divided by anything, the remainder goes into accumulator 0.

To print out numbers between ten and ninety-nine, what we must do, therefore, is divide by ten. Then we print out the quotient, followed by the remainder. Remember that the number ten, by which we want to divide, is to be entered in our program as an octal number; D 10 = O 12.

```
        INT=1
        REM=2
START:  INCHWL  INT
        SUBI    INT,60
        IMULI   INT,2
PRINT:  IDIVI   INT,12
        ADDI    INT,60
        OUTCHR  INT
        ADDI    REM,60
        OUTCHR  REM
        EXIT
        END     START
```

This program uses two accumulators. Division is carried out on the contents of accumulator 1, so the remainder appears in accumulator 2; hence our choice of the name REM for accumulator 2.

The label PRINT could be omitted; it serves no function in the program. We have included it solely to mark the point at which, with calculation completed, the process of printing out begins.

Experiment with this program, and see that it will double numbers up to and including 9; although now if we input, say, 4, the result of doubling is printed out as 08 (why?). We still cannot input larger numbers.

Now we shall extend the program in another direction. Instead of multiplying always by 2, we shall choose the number by which to multiply when the program runs. In effect, our program will now form the product of two single-digit numbers.

We shall need another accumulator to receive the second number. Let us use accumulator 3, and give it the name NUM . When we have put our numbers into INT and NUM, we want to multiply the contents of these two accumulators. The instruction for this is

```
IMUL        INT,NUM
```

and the product goes into INT, because INT is the one that comes before the comma.

The difference between IMUL and IMULI is crucial. Compare

```
(a) IMULI   INT,2
(b) IMUL    INT,2
```

(a) multiplies the contents of the accumulator named INT by the *number 2*

(b) multiplies the contents of the accumulator named INT by the *contents of accumulator number* 2.

In both cases, the result of the multiplication is put in accumulator INT. In case (b) the contents of accumulator number 2 are unaffected (unless in our program we have set INT=2, in which case the result is to multiply the contents of INT by themselves; in other words, to square the contents of INT).

The following program now takes in our two numbers, multiplies them together, and prints out the result. When you EXecute this program (having chosen a name for it and created it with TECO), the machine will wait until you type in two numbers followed by a ↵, then print out the product and exit.

```
        INT=1
        REM=2
        NUM=3
START:  INCHWL  INT
        INCHWL  NUM
        SUBI    INT,60
        SUBI    NUM,60
        IMUL    INT,NUM
PRINT:  IDIVI   INT,12
        ADDI    INT,60
        OUTCHR  INT
        ADDI    REM,60
        OUTCHR  REM
        EXIT
        END     START
```

If you type in number–space–number, you will not get the correct result. (Why not?)

The instruction for adding the contents of two accumulators is ADD. You should amend the above program, to make it add two numbers together, by changing

```
IMUL        INT,NUM
```

to

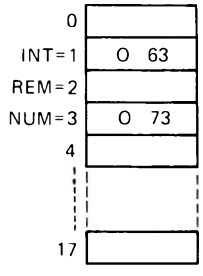
```
ADD         INT,NUM
```

Observe that the distinction between ADD and ADDI is analogous to that between IMUL and IMULI.

Separator Characters and Skip Instructions

Let us return to the question raised above. Suppose that we execute the above program, with the desire to multiply 3 by 8. If we type 38↵ then 24 correctly appears. But if we type 3;↵, with a semicolon used to separate 3 and 8, the result is 33, plus, after exiting, a very enlightening message from the monitor (Try it!). Other separating characters will produce various exotic results.

We can follow through what happens line by line of the program. The first instruction received is INCHWL INT . Because INCHWL is a "wait on line" instruction, the machine can do nothing until a ↵ is received. Then it has the characters 3 followed by ; then 8 and the two characters comprising ↵ in its "buffer" ready to be processed. Now the instruction INCHWL INT has the first character, in the form of its ASCII code, placed in INT. The first character is 3 and its ASCII code is O 63, so O 63 goes into INT. The next instruction is INCHWL NUM . Now the next character in the buffer is ; and its ASCII code happens to be O 73; this number is placed in accumulator NUM.



The next two lines, subtracting O 60 from each, leave us with O 3 in INT, O 13 in NUM. Now O 13 = D 8 + 3 = D 11, so the result 33 is "correct." The program will reach its end without having the character 8 in the buffer ever reached; it remains there to puzzle the monitor.

As we proceed, it will become plain that the use of assembler language gives precise and total control over the operations of the computer. For the present, however, the process of actually exercising that control may appear burdensome, and the rewards nonexistent. As we progress, this balance will gradually change in our favor.

We must find a way to enable the computer to recognize separator characters in our input. This is absolutely necessary for even trivial problems. Suppose we extend our last program to multiply together two numbers of any size. Then somehow the machine must distinguish input of 35 and 62 from input of 3 and 562. This depends on where the separator character occurs.

To begin with, we shall separate our numbers by a ↵. Our program must recognize that ↵ indicates that input of a number has just ended. It must also refrain from treating the ↵ itself as if its characters were the next two characters of the input data.

Now ↵ comprises the two characters:

CARRIAGE RETURN—ASCII code O 15

LINE FEED—ASCII code O 12

We shall prepare the ground for input of a number of more than one digit. We shall input a character and check whether it is a carriage return. If so, we ignore it, and the following line feed, and input the next character.

This brings us to the most far-reaching power of the computer: the ability to take alternative courses of action, depending on the result of a previous step. This is usually achieved by an instruction that compares two quantities, and depending on the comparison, either skips over the following line in the program or does not.

We first consider a class of instructions that compare the contents of an accumulator with an actual number; these are four or five letter codes, of which the first three letters are CAI—acronym

for Compare Accumulator Immediate. The remaining letters give the circumstances under which the next instruction in the program is to be skipped. We have

```
CAIL        accumulator,number
```

which means: skip if the contents of the accumulator are less than the number. Similarly, CAI can be followed by

LE	less than or equal to
G	greater than
GE	greater than or equal to
E	equal to
N	not equal to

Thus, for example, the instruction

```
CAIG        INT,71
```

will cause a line skip if accumulator INT contains a number greater than O 71, and not otherwise.

The following amended form of our program enables the two single-digit inputs to be separated by a ↵.

```
        INT=1
        REM=2
        NUM=3
START:  INCHWL  INT
        INCHWL  NUM
        CAIN    NUM,15
        INCHWL  NUM
        CAIN    NUM,12
        INCHWL  NUM
        SUBI    INT,60
```

and so on, as before.

Let us follow through carefully what happens. The first character goes into INT. The next character goes into NUM, and is then compared with 15, the ASCII code of carriage return. If our character was not a carriage return, then NUM does not contain 15, so we skip. If NUM does contain 15, we do not skip, and the next line tells the monitor to move on to the next character, and take it into NUM. The procedure is now repeated, this time so as to exclude 12 (line feed) as well as 15.

Note what happens on the instruction INCHWL NUM , when accumulator NUM already contains something. The previous contents are discarded, and replaced with the next character in line for input.

Our program still suffers from the inelegance of printing out 2 times 4 as 08. This is because the instruction OUTCHR INT is carried out even if INT (which here contains the number in the tens column) is zero. We can suppress this leading zero by first comparing the contents of INT with O 60, the ASCII code for 0, and skipping the instruction to print out in case of equality. So the printing routine becomes

```
PRINT:  IDIVI   INT,12
        ADDI    INT,60
        CAIE    INT,60
        OUTCHR  INT
```

and so on. Of course we do not want to suppress the printing out of a zero from REM as well! (Why not?)

To print out numbers of possibly more than two digits, the process of dividing by ten and saving the remainder must be repeated the appropriate number of times. Suppose we know that the contents of INT may be a number of at most four digits; in that case, we must divide by ten three times over, then print out the contents of INT followed by the three remainders (suppressing INT

itself if it contains zero). For example, if INT contains 2174, successive division by ten gives

	INT	REMAINDERS		
	2174			
after 1st division	217	4		
after 2nd division	21	7	4	
after 3rd division	2	1	7	4

We will need four accumulators: INT itself, and three for the remainders that give the hundreds, tens, and units. Let us call these three HREM, TREM, UREM. We shall begin by designating

```
INT=1
HREM=2
TREM=3
UREM=4
```

If, initially, we suppose that INT contains 2174, then

```
IDIVI   INT,12
```

puts 217 in INT, and 4 in the next accumulator, HREM. We must move the contents of HREM into UREM. This is done by

```
MOVE    UREM,HREM
```

Note the order! Note also that the contents of HREM are not changed by this instruction. The whole print routine for a four-digit number now looks like this:

```
PRINT:  IDIVI   INT,12
        MOVE    UREM,HREM
        IDIVI   INT,12
        MOVE    TREM,HREM
        IDIVI   INT,12
        ADDI    INT,60
        CAIE    INT,60
        OUTCHR  INT
        ADDI    HREM,60
        OUTCHR  HREM
        ADDI    TREM,60
        OUTCHR  TREM
        ADDI    UREM,60
        OUTCHR  UREM
        EXIT
        END     START
```

This will print out four-digit numbers correctly, and also three-digit numbers, since it suppresses the leading zero. It will, however, print out 27 as 027, 3 as 003, and zero as 000. At present it would be complicated to correct this stylistic defect. It is a good exercise to endeavor to do so; it will familiarize you with the problems involved. Observe that a program that prints 27 as such, instead of 027, is not much use if as a result it prints 1027 as 127.

Leaving this point for later, let us consider how to input a two-digit number. We have already managed to mark the end of the input of a number by a carriage return. So we can proceed as follows, using, for example, 26. We type in 26↵. The machine takes 2 and 6 into different accumulators; multiplies 2 by ten and adds 6 to the result. This procedure will work if the input is a single digit, as long as we are careful not to put it into the "tens" accumulator. The following routine puts a one- or two-digit number correctly into INT. HREM is accumulator 2, as designated above.

```
INCHWL  INT
INCHWL  HREM
SUBI    INT,60
SUBI    HREM,60
CAIL    HREM,0
IMULI   INT,12
CAIL    HREM,0
ADD     INT,HREM
```

The difficulty in writing this was to ensure that both one- and two-digit numbers are interpreted correctly. Let us follow through the above routine for both cases. First, suppose we try to input twenty-seven; so we type 27↵. Then the ASCII code for 2, which is O 62, goes into INT; similarly, O 67 goes into HREM. Subtraction of O 60 now puts 2 in INT, 7 in HREM.

The next line compares the contents of HREM with zero, and will skip if HREM contains a quantity less than zero (we shall see the purpose of this step below). Since HREM contains 7, the program does not skip. The next instruction multiplies the contents of INT by O 12; that is, by ten. INT now contains twenty, HREM still contains 7. The next skip instruction is the same as the previous one, and is not effective in this case. So the ADD instruction is carried out. The contents of HREM are added to the contents of INT, which now, as desired, contains twenty-seven.

Accumulator HREM still contains 7. This will be obliterated later when we divide the contents of INT by ten, which puts the remainder into HREM in place of any former contents.

Now suppose we want to input five; so we type 5↵. Then the ASCII code for 5, which is O 65, goes into INT. This time, however, HREM receives O 15, the ASCII code for carriage return. After subtraction of O 60, INT contains 5; HREM contains a negative number. (It is in fact O −43, but there is absolutely no reason to perform this octal calculation. It is quite enough to observe that it must be negative.)

This time the skip instruction does take effect; the skip takes us to the next skip instruction, which is again effective, and the program skips to whatever follows the above routine. Thus we finish with INT containing 5. As before, we are not now concerned with the contents of HREM.

```
        INT=1
        HREM=2
        TREM=3
        UREM=4

START:  INCHWL  INT             ;(a)
        INCHWL  HREM
        SUBI    INT,60
        SUBI    HREM,60
        CAIL    HREM,0
        IMULI   INT,12
        CAIL    HREM,0
        ADD     INT,HREM

        INCHWL  TREM            ;(b)
        CAIN    TREM,15
        INCHWL  TREM
        CAIN    TREM,12

        INCHWL  TREM            ;(c)
        INCHWL  UREM
        SUBI    TREM,60
        SUBI    UREM,60
        CAIL    UREM,0
        IMULI   TREM,12
        CAIL    UREM,0
        ADD     TREM,UREM

        IMUL    INT,TREM        ;(d)

PRINT:  IDIVI   INT,12          ;(e)
        MOVE    UREM,HREM
        IDIVI   INT,12
        MOVE    TREM,HREM
        IDIVI   INT,12
        ADDI    INT,60
        CAIE    INT,60
        OUTCHR  INT
        ADDI    HREM,60
        OUTCHR  HREM
        ADDI    TREM,60
        OUTCHR  TREM
        ADDI    UREM,60
        OUTCHR  UREM
        EXIT
        END     START
```

FIGURE 2.1 A program to multiply two one- or two-digit numbers.

We conclude this section with a program that will multiply together any two numbers of one or two digits each. When the program is executed, the numbers are entered with a ↵ after each of them. The program is in Figure 2.1.

Input of the first number is handled using accumulators INT and HREM, with the number finally reaching INT. For the second number, we use TREM and UREM for its digits, finally getting the number into TREM. Although we chose these names with printout in mind, they do not restrict the use to which we can put the accumulators.

Now we multiply the contents of INT by the contents of HREM. Finally, we use our printout routine on the new contents of INT.

It is vital for your progress that you study this program until you fully understand the effect of each single step. Make out a table with four columns headed INT, HREM, TREM, UREM. Take some particular examples for input, and work through the program line by line. In successive lines of your table, write in what the contents of the four accumulators will be after the corresponding line of the program has been reached. Doing this exercise thoroughly is very beneficial later on.

After you have done this, try putting these notes aside and constructing the program again for yourself. Does it run properly for various choices of input? If not, create a table again for your program, and check through, line by line, what happens to a given input. When you have corrected all the errors you can find (using TECO), execute the program again. If it still does not work, repeat the process until it does! Learning to tolerate patiently the tedious chore of "debugging" is one of the least enjoyable, but regrettably one of the most essential aspects of programming.

When you have done this, you might refer to Appendix A on debugging, and try it over again using DDT.

Notes on Figure 2.1:
(a) Input first number
(b) Discard carriage return and line feed separating numbers
(c) Input second number
(d) Form product
(e) Output product

Comments

In longer programs it can be helpful to include brief notes explaining the purpose of a line or a collection of lines. This is particularly useful if programs written by one person are to be read by another. To include comment on a line, precede the comment by a semicolon, just as we did above. A whole line of comment must begin with a semicolon as its first-nonspacing character. The above program might include

```
        ; now follows the printout routine
PRINT:  IDIVI   INT,12   ;O 12 = D 10
```

How much comment should be included is to some extent a matter of individual taste. Few programmers would include as much as in the above example. After all, choosing the label PRINT obviates further comment on the purpose of the routine. Some would include the comment on division by O 12; for others, such a frequently occurring line requires no comment. Certainly the comment in such a line as

```
PRINT:  IDIVI   INT,10   ;octal printout
```

is worthwhile; otherwise on later reading, by the programmer who wrote it or anyone else, the natural assumption would be that a blunder had been made. In general, lines of comment should indicate program flow from one stage to another. Individual instructions deserve a comment if their function is not fairly clear, and most certainly if any subtle trickery is involved. It can also be

helpful to explain accumulator usage:

```
CT=1          ;character count
```

Although you should develop your own style regarding comments, be sure it conforms with the general principles we have outlined.

We have purposely been very sparing with comments in several of the programs in this book, particularly in the early stages. These programs are exercises as well as illustrations. Your first approach to each of them should be careful line by line study, writing comments where apt for what is clear to you, reserving queries for any instruction whose purpose you cannot fathom. Then, and not before, copy the program as your own file, and work through it using DDT. Try various inputs, and see that the program does what it should. Do not be satisfied until you understand the function of every single line. Finally, make and keep a copy of the program that is fully annotated with your own comments. This approach will rapidly develop your own program writing skills.

Exercises: Write a program that . . .

(i) accepts input of two numbers of up to two digits each, and prints out
  (a) the larger of them;
  (b) the (positive) difference between them;
  (c) the smaller, a semicolon, then the greater.
  Be sure that your program can cope when the numbers are equal.

(ii) accepts input of a number of up to three digits followed by a single digit number, and prints out the quotient when the former is divided by the latter. Have your program just EXIT if the divisor is zero.

(iii) accepts input of two octal numbers of up to two digits each, and prints out their product as an octal number. Have your program exit if the numerals 8 or 9 appear in the input.

(iv) accepts input of an octal number of up to four digits, and prints it out as a decimal number.

(v) the opposite of (iv).

2.2 JUMP INSTRUCTIONS

The computer carries out the instructions in a program successively, line by line. In the last section we learned the CAI- instructions, which cause this sequential mode of operation to be changed. Depending on the result of a certain comparison, the instruction next following may be passed over. But this hardly helps us if we want the carrying out of a whole routine to depend on a certain comparison of quantities—a frequent need in programming. Consider the section marked (a) in the program at the end of Section 2.1, and see the clumsy way in which we managed to make the performance of the two instructions `IMULI INT,12` and `ADD INT,HREM` depend on the contents of HREM. Such matters are handled more elegantly using an instruction to jump to another point in the program. The usual format of instructions is

```
skip depending on a comparison
jump to appropriate point
```

so that whatever instruction follows this fragment is carried out in case of a skip. Otherwise, the jump instruction takes effect, and some special routine, to be found elsewhere in the program, is performed. The conclusion of this routine might be a jump instruction returning us to the next instruction after the point of departure.

We introduce the jump instruction JRST. A label at the beginning of the destination line is used to complete the instruction. The label itself is always followed by a colon, but reference to it in

the jump instruction must not include the colon. Thus, incorporating the instruction

```
        JRST    PRINT
```

would cause a jump to the line labeled PRINT

```
PRINT:  IDIVI   INT,12
```

in the last program of Section 2.1. It does not matter whether the jump instruction is placed in the program before or after the destination of the jump.

As an example of how availability of the jump instruction increases our programming capabilities, we shall write a more general routine to input a number. We want to be able to type in a number of any size, in the usual decimal notation, and finish with a ↵. So the number of digits to be entered is not predetermined. Our routine will take in the number, digit by digit. At each stage, if the character taken in is a carriage return, then input is finished, and the number already stored is what is wanted. Otherwise, the latest digit is added to ten times the number already stored. Let us examine this process with an example (in decimal notation), say, 234. Input is successively 2, 3, 4, ↵. First, 2 is stored. Since 3 is seen to be the next digit, we form $(2 \times 10) + 3 = 23$. The next digit is 4, so we form $(23 \times 10) + 4 = 234$. There follows the ↵, so we are done. Of course, our routine must convert from ASCII codes to the corresponding numbers. Here is such an input routine. The first command SETZM sets the contents of the stated location to zero. We then take characters into accumulator DGT. On finding a carriage return, we jump to some line elsewhere in our program; the line must bear the label DONE . Otherwise, we know that DGT contains the next digit, and we proceed as indicated above. You should work through this routine carefully, line by line, for various choices of input.

```
        INT=1
        DGT=2
        . . .
        SETZM   INT

LABEL:  INCHWL  DGT
        CAIN    DGT,15
        JRST    DONE
        SUBI    DGT,60
        IMULI   INT,12
        ADD     INT,DGT
        JRST    LABEL
```

This routine will input any whole number not too large to be contained in a single word of the computer. The 36 bits (this is D 36) of a computer word are numbered 0 through 35, and all but bit 0 can be used in the representation of a positive whole number. It turns out that this permits holding all decimal numbers of up to ten digits. (Exercise for the reader with some mathematical knowledge: what precisely is the largest integer that can be held in a single computer word?)

If you try to input too large a number, you will not set an error message, but your results will be incorrect. Output of numbers comprising varying numbers of digits is somewhat more difficult. We have to divide by ten, and store the successive remainders. When division by ten has reduced the original number to zero, we print out the remainders, starting with the last and ending with the first. You should confirm this method by trying it on a few examples. The trouble is that since we do not know how large the number to be output may be, we cannot anticipate the number of accumulators needed for the successive remainders.

Indexing

We can overcome this by *indexing*. We can set aside one accumulator—let us call it N—for indexing. This may be any accumulator *except accumulator number 0*. Accumulator N will never hold a remainder; rather, it will hold the number of the accumulator into which a particular remainder is to go. For example, suppose in the accumulator called REM we have a number we want to put in accumulator 7. Then we first make sure that the contents of N are set equal to the number 7. This

is achieved by a MOVE Immediate instruction

```
MOVEI    N,7
```

and now

```
MOVEM    REM,(N)
```

puts the contents of REM into accumulator 7. There are two new things in this last instruction. MOVEM is similar to MOVE, except that it goes in the opposite direction, moving the contents of the location on the left to the one on the right; we shall have more to say about this in the next section. The notation (N) causes the contents of REM to be moved, not to accumulator N itself, but to the accumulator whose number is given by the contents of accumulator N. Distinguish carefully between

```
(a) MOVEM  REM,N
(b) MOVEM  REM,(N)
```

(a) moves the contents of the accumulator named REM into the accumulator named N
(b) moves the contents of the accumulator named REM into the accumulator *whose number is given by the contents* of N.

In each case, the contents of REM are unchanged.

Of course MOVE 7,REM or MOVEM REM,7 would each be a simpler way to do this. The power of indexing, however, lies in our ability to increase the contents of N at each successive step, stringing out the successive remainders in sequence.

We shall use accumulator 1 for the number to be printed out. On division by ten, the remainder gets put into accumulator 2. Accumulator N=3 will serve for indexing. This leaves, for holding the successive remainders, accumulators 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17; twelve in all, which is more than enough for the decimal digits corresponding to the contents of a single accumulator word.

We start by setting the contents of accumulator N equal to 3. Thus, accumulator number 3 also contains the number 3. Since N is to be used as a sort of *pointer,* we can think of it as starting off pointing to itself. Each time we divide by ten, we increase the contents of N by 1, so that N points to the next accumulator; and into that accumulator we put the remainder from the division. We repeat the process until all remainders are found. Accumulator N now points to the last remainder found: so we print it out, and decrease the contents of N by 1. This process is repeated until, finally, the contents of accumulator 4 are printed out. Note how carefully we must ensure that accumulator N points to exactly the right place: it is all too easy to be inaccurate in this by one place. (What happens if we start off with N containing 4, the first location to be used for holding remainders? Rewrite the program below, starting in this way.) Here is such an output routine:

```
        INT=1
        REM=2
        N=3
        . . .
        MOVEI   N,3
L1:     IDIVI   INT,12
        ADDI    N,1
        ADDI    REM,60
        MOVEM   REM,(N)
        CAIE    INT,0
        JRST    L1

L2:     CAIGE   N,4
        JRST    DONE
        OUTCHR  (N)
        SUBI    N,1
        JRST    L2
```

You should pay particularly careful attention to this routine; study it with the aid of several numerical examples.

A Multiplication Program

Let us use our input and output routines to write a complete program (Figure 2.2). We shall write a multiplication program more general than that of Section 2.1, in that any two whole numbers can be multiplied together. If the result of multiplication is too large to be held in one word, this program will produce incorrect printout.

Notice how we keep a *block structure* of separate *routines.* Once we know how to perform a certain kind of operation, we can carry the routine for it virtually intact from one program to another.

We leave blank lines to stress the block structure. Although TECO is aware of a blank line as being a line, such lines are wholly ignored when your program is run. In the above routine, if INT contains zero the instruction CAIE INT,0 will cause a skip to the line bearing the label L2.

Instead of letting our program exit after just one multiplication, we jump from the printout routine back to the start. At this point we put in a carriage return and two line feeds for a pleasing format. As a refinement, we have the symbol ? (ASCII code O 77) printed out whenever the program is waiting for input. So, on executing the program, wait for a ? then input the first number followed by ↵. A second ? will appear, and you then type in the second number followed by ↵. The product will now appear, and the whole process will start again.

Since this program never reaches its END statement, there is no need for an EXIT instruction (refer to Section 1.3). To escape from the program, press ^C; if the machine is actually calculating when you do so, you will need a further ^C.

Observe how we enable the program to expect input of precisely two numbers. Our input

```
        INT=1
        REM=2
        NUM=3

START:  SETZM
L0:     SETZM   INT
        OUTCHR  [77]
L1:     INCHWL  REM
        CAIN    REM,15
        JRST    L2
        SUBI    REM,60
        IMULI   INT,12
        ADD     INT,REM
        JRST    L1

L2:     CAIE    0
        JRST    L3
        MOVE    NUM,INT
        ADDI    1
        INCHWL  REM
        JRST    L0

L3:     IMUL    INT,NUM

PRINT:  MOVEI   NUM,3
P1:     IDIVI   INT,12
        ADDI    NUM,1
        ADDI    REM,60
        MOVEM   REM,(NUM)
        CAIE    INT,0
        JRST    P1

P2:     CAIGE   NUM,4
        JRST    REPEAT
        OUTCHR  (NUM)
        SUBI    NUM,1
        JRST    P2

REPEAT: OUTCHR  [15]
        OUTCHR  [12]
        OUTCHR  [12]
        INCHWL  REM
        JRST    START
        END     START
```

FIGURE 2.2 A program to multiply any two numbers.

routine puts a number into INT. We move the first input from INT to NUM, then repeat the input routine. So we end up with our two numbers in INT and NUM. We make sure that there is no attempt to carry out the input routine more than twice by using accumulator 0 as a counter. (What would happen if we did not take this precaution?) Work out for yourself how this is done, remembering that instructions referencing an accumulator, but not mentioning any one specifically, refer to accumulator 0. Thus CAIE 0 causes a skip if the contents of accumulator 0 are equal to zero; and ADDI 1 adds 1 to the contents of accumulator 0.

The purpose of the INCHWL REM instructions to be found in routines L2 and REPEAT is to dispense with the line feeds between and after the two numbers. How does this work? And why is it necessary?

Notes on Figure 2.2:

L1: This routine puts into INT a number typed at the terminal in normal decimal notation, and followed by ↵. The contents of INT must be zero at the start of this routine. It may be described as a routine to *read* a number.

L2: This routine transfers the contents of INT to NUM; sets INT to contain zero; disposes of the line feed between the two numbers being input; and uses accumulator 0 to ensure that this routine is carried out exactly once, so that just two numbers are read.

L3: This one line is the whole arithmetical calculation!

PRINT: The print routine was examined previously.

REPEAT: Formats ready for input of further numbers.

Exercises: Write a program that . . .

(i) reads a number and prints out its square;

(ii) reads a number and prints out its cube;

(iii) reads a number n and prints out $n!$ where $n! = n(n-1)(n-2)...2.1$;

(iv) reads two numbers and prints out the remainder when the larger is divided by the smaller;

(v) reads two numbers m and n, and prints out the nth power of m.

Include in your programs any comments you consider suitable.

Counting Data Items

In the above examples, the number of items of data was known in advance. As an example of how to escape this restriction, we shall construct a program to read a collection of numbers and compute their mean (average). This program is in Figure 2.3.

To calculate the mean, we will need to know how many data items were input. This is done by using an accumulator as counter, increasing its contents by 1 every time a number is read.

It is not a good idea to terminate the entire input with ↵, since input may need to extend beyond just one line of type. As convenient a system as any is to use $ (ESCAPE—ASCII code O 33) to signal the end of all data input, and to let any other nonnumeric character serve as a separator between numbers. Recall that numerals have ASCII codes O 60 through 71, so it is a simple matter to check if a character is a numeral or not.

The program must take care of the possibility of more than one separator character between numbers, or of a separator character before the terminating $. Otherwise, excessive care in typing will be required at execution time. So, on finding a separator character, the program must go to a routine that discards any further separator characters, before attempting to read the next data item.

To begin, we set out counter CT to contain 0. Our READ routine reads a number, using accumulators INT and DGT. On finding a separator character, routine SEP increases the contents of CT by 1; adds the contents of INT to the running total held in NUM; resets the contents of INT to zero in preparation for reading the next data item; discards any further separator characters; and, if a numeral turns up, returns it to the appropriate point in READ (returning to the start of

```
        CT=0
        INT=1
        DGT=2
        NUM=3
        REM=4

START:  SETZM   CT
        SETZM   INT
        SETZM   NUM

READ:   INCHWL  DGT
        CAIGE   DGT,60
        JRST    SEP
        CAILE   DGT,71
        JRST    SEP
R1:     SUBI    DGT,60
        IMULI   INT,12
        ADD     INT,DGT
        JRST    READ

SEP:    ADDI    CT,1
        ADD     NUM,INT
        SETZM   INT
S1:     CAIN    DGT,33
        JRST    MEAN
        INCHWL  DGT
        CAIGE   DGT,60
        JRST    S1
        CAILE   DGT,71
        JRST    S1
        JRST    R1

MEAN:   IDIV    NUM,CT
        IMULI   REM,2
        SUB     CT,REM
        CAIG    CT,0
        ADDI    NUM,1
        JRST    PRINT

PRINT:  supply for yourself a routine to
        print out the contents of NUM.

Then finish the program.
```

FIGURE 2.3 A program to compute the mean of a collection of numbers.

READ would lose the character!); if $ turns up, the program jumps to MEAN . The process is illustrated by a flow chart in Figure 2.4.

The mean is calculated to the nearest whole number; analyze for yourself how this is achieved. The jump instructions of SEP should be studied with especial care.

Notes:
SUB CT,REM subtracts the contents of REM from the contents of CT. The contents of REM are unchanged.

Observe how we "round up" the result if the remainder indicates that a fractional part of one half or more has been lost.

Exercise: Write a program to read a collection of numbers and print out the least and the greatest of them.

SECTION 2.3 MEMORY

In Section 2.2 we considered the problem of reading in a large number of items of data; this was in the program that calculated the mean of a collection of numbers. We could do this with only the sixteen accumulators at our disposal simply because we did not need to store all the data items separately; we totaled as we went along. Not all processes, however, can be dealt with in such a

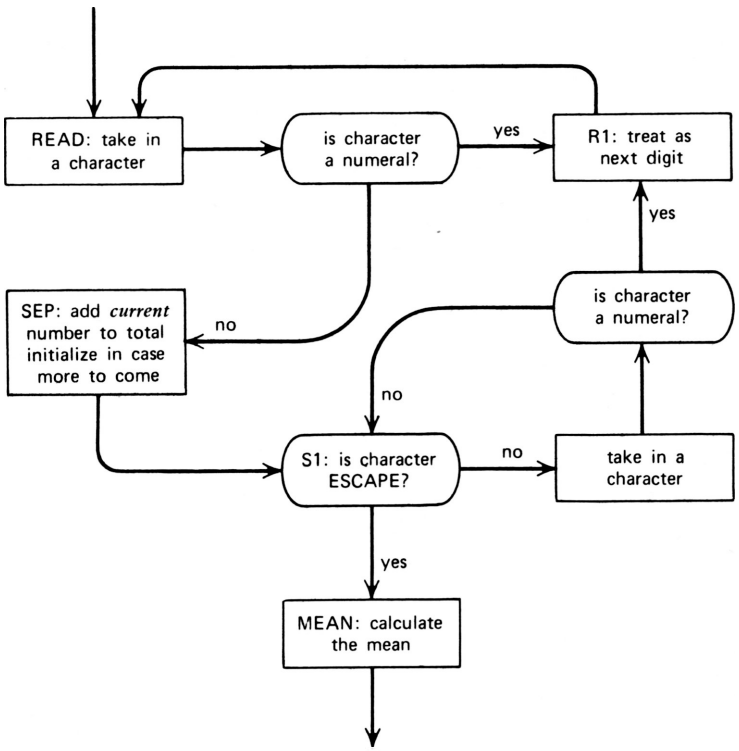


FIGURE 2.4 Flow chart to accompany the program in Figure 2.3.

way, and much of the power of the computer depends on its ability to store and retrieve large quantities of data. To this end, *memory* (or *core*) is available. Memory consists of a large number of computer words accessible to the programmer. It should be mentioned at the outset that memory space is needed by many users at a time; so the amount available to each individual user, although considerable, is not unlimited, and should not be unduly wasted.

The sixteen accumulators available to each user are themselves memory locations, but are rather special ones. There are many operations that can be carried out on the contents of an accumulator, but not on any other memory word.

The memory space needed by your program must be specifically claimed by it. Single words can be declared by the special symbol # as the program proceeds. Suppose we want to retain the contents of accumulator AC for later use in our program. We must invent our own name for the memory word we want to use; let us call it MEM. In one line we can declare it as such, and using the MOVE to Memory instruction MOVEM we can deposit the contents of AC in MEM (the contents of AC remain unchanged):

```
MOVEM     AC,MEM#
```

When the program is assembled, # indicates that the next available location is to be reserved as MEM. Thus, # should appear only on the first use of MEM in the program. There must be no space between MEM and #.

AC and MEM are perfectly good designators for actual program use; but because of their mnemonic qualities, we shall use them to indicate, in describing an instruction, that some accumulator and some memory word are involved. The computer manuals often use ADR (mnemonic for address) or E to indicate that a memory location is involved.

MOVEM is one of a host of instructions that are to be followed by accumulator and memory addresses, in that order, and separated by a comma. Spaces after the comma are acceptable. MOVE is another such instruction. In Section 2.1 the memory locations were all accumulators, but now we

must be aware that

```
MOVE    AC,MEM
```

moves the contents of MEM into AC. Whatever was formerly in AC is lost; the contents of MEM are unaffected.

```
MOVEM   AC,MEM
```

goes in the opposite direction. Avoid

```
MOVE    MEM,AC
```

it will not work (unless MEM happens to be an accumulator).

If WORD is any memory location, we will indicate its contents by parentheses: (WORD). Using this notation, we can indicate the effects of MOVE and MOVEM like this:

MOVE	AC,MEM	(MEM) ———→ (AC) (MEM) unchanged
MOVEM	AC,MEM	(AC) ———→ (MEM) (AC) unchanged

In the group of MOVE- instructions, we have also

MOVEI	AC,X	X ———→ (AC)

in which X denotes an actual number.

In Section 2.1 we learned the Compare Accumulator Immediate instructions:

CAIE	AC,X	if (AC) = X, skip
CAIN	AC,X	if (AC) ≠ X, skip
CAIG	AC,X	if (AC) > X, skip

and so on for CAIGE, CAIL, CAILE.

Analogously we have Compare Accumulator with Memory instructions:

CAME	AC,MEM	if (AC) = (MEM), skip
CAMN	AC,MEM	if (AC) ≠ (MEM), skip
CAMG	AC,MEM	if (AC) > (MEM), skip

and so on for CAMGE, CAML, CAMLE. The instructions CAIA and CAMA Always skip; CAI and CAM alone *never* skip, so these instructions do nothing at all.

The various arithmetical operations are

ADD	AC,MEM	(AC) + (MEM) ———→ (AC) (MEM) unchanged
ADDM	AC,MEM	(AC) + (MEM) ———→ (MEM) (AC) unchanged
ADDI	AC,X	(AC) + X ———→ (AC)
IMUL	AC,MEM	(AC) × (MEM) ———→ (AC) (MEM) unchanged
IMULM	AC,MEM	(AC) × (MEM) ———→ (MEM) (AC) unchanged
IMULI	AC,X	(AC) × X ———→ (AC)

The subtraction and division processes require care: subtraction is always subtraction of (MEM) from (AC), and division is always division by (MEM) into (AC).

SUB	AC,MEM	(AC) − (MEM) ———→ (AC) (MEM) unchanged
SUBM	AC,MEM	(AC) − (MEM) ———→ (MEM) (AC) unchanged
SUBI	AC,X	(AC) − X ———→ (AC)
IDIV	AC,MEM	(AC) / (MEM) ———→ (AC) (MEM) unchanged remainder ———→ AC+1
IDIVM	AC,MEM	(AC) / (MEM) ———→ (MEM) (AC) unchanged remainder lost
IDIVI	AC,X	(AC) / X ———→ (AC) remainder ———→ AC+1

Exercises: Write routines to
(i) interchange the contents of two memory locations;
(ii) divide the contents of a memory location by the contents of an accumulator.

Lists

Data with some inherent ordering clearly needs to be retrievable in the order in which it is input, otherwise essential information may be lost. A reasonable approach is to put consecutive items into consecutive memory locations. The location after MEM can be referred to simply as MEM+1. Next follows MEM+2, and so on. These numbers are octal, so after MEM+7 we have MEM+10. There must be no spaces on either side of the + sign. This method of reference is adequate for a few locations, but it does not give us a general way to refer to individual locations in a large block of memory words. To do this we use the full power of *indexing*. The notation

MEM(AC)

means the memory location X locations after MEM, where X is the number contained in AC. AC may be any accumulator except number 0.

Of course MEM may be an accumulator. If it is accumulator 0 we may suppress specific reference to it. So (AC) will mean: X locations after accumulator 0, where X is the number contained in AC. If X is between 0 and O 17, then (AC) is just accumulator number X. This is the way we introduced indexing in Section 2.2.

The location after MEM(AC) may be referred to as MEM+1(AC), but *not* as MEM(AC)+1; indexing must come after the addition. (Why is MEM(AC+1) not necessarily the next location after MEM(AC)? Under what circumstances would it be?)

To ensure that a block of memory locations will be available, we must reserve it. To do so, we put into our program—after the instruction to be performed but before the END statement—a line such as

```
MEM:    BLOCK    1000
```

This demands a block of O 1000 locations, starting from some location in the core, which will be recognized by the name MEM in your program. The locations are, therefore, MEM through MEM+777. As a guide to how much memory space to claim, be aware that

O 1000 = D 512
O 10 000 = D 4 096

If you cannot be sure exactly how much memory space your program will need, declare a block large enough to allow a margin of safety; but, in consideration of other users, please do not be overly extravagant.

A BLOCK declaration takes the place of individual word declarations. So if you are going to declare a block with MEM in it, do not write # next to MEM when you introduce it in the body of your program. Note that the name MEM is our choice, but BLOCK is assembler language terminology.

It may at times be necessary to ensure that certain individual memory locations are consecutive; this will not occur automatically unless they make their first appearances in the program consecutively. Otherwise, we can make declarations, for individual locations as for blocks, after the end of instructions and before the END statement. For example, if we want MEM, ABL, and WRD to occupy consecutive locations, we do not introduce them in the program with #, but rather declare them before the END statement like this

```
MEM:    0
ABL:    0
WRD:    0
```

The effect of the 0 is to set the contents of the word equal to 0 when the program is assembled; that is, it declares an initial value for the contents of the word. If we wanted ABL initially to contain the number ten, we would arrange this by

```
ABL:    12          ;O 12 = D 10
```

which saves us the trouble of having

```
        MOVEI   AC,12
        MOVEM   AC,ABL
```

within the body of our program.

Sorting

To illustrate the use of blocks of memory locations, we shall write a program to read a list of whole numbers, then print them out in increasing numerical order. This calls for a rearrangement of the data items into numerical order. Such programming serves as a good introduction to the frequently occurring problem of arranging a list of words in alphabetical order. There are several different methods available for this *sorting* of data; which is most efficient will depend on the circumstances. Our choice of method is quite efficient, as well as fairly straightforward to program.

To arrange our list of numbers in increasing numerical order, think of them as one long list across a page. Starting from the left, we move across the list to the right. On reaching each number, we compare it with its successor to the right. If the successor is greater or equal, we move on one step to the right. Otherwise, we interchange the two numbers, and move one step to the left to see if further exchanges of the smaller number are needed (if we are already at the leftmost number, we carry on to the right). We proceed until we reach the rightmost number.

For example, starting with

```
2   1   7   5   3
```

the successive steps are

```
1 # 2   7   5   3
1   2 * 7   5   3
1   2   5 # 7   3
1   2 * 5   7   3
1   2   5 * 7   3
1   2   5   3 # 7
1   2   3 # 5   7
1   2 * 3   5   7
1   2   3 * 5   7
1   2   3   5 * 7
```

On each line we have placed, between the two numbers that have just been compared, a # if they have been interchanged, a * if not. As you see, when we can compare the last two numbers and find it unnecessary to interchange them, we are finished.

This may seem rather complicated, but the heart of the program is a simple routine COMPAR to compare two numbers and possibly interchange them. Suppose we have our data numbers contained in locations MEM through MEM+X, where X is a number contained in accumulator AC. X is, of course, the number of data items, less 1. (Why "less 1"?)

We will use accumulator N to keep track of where we are in the list. We can refer to any item in the list by the indexed address MEM(N), as long as we have put into N the appropriate number between 0 and X. We shall, therefore, be comparing the contents of MEM(N) with the contents of MEM+1(N). Neither of these is an accumulator, so we cannot compare them in one instruction. Instead, we move the contents of MEM(N) into accumulator 0, then compare the contents of accumulator 0 with those of MEM+1(N). If the former is greater than the latter, we jump to the SWAP routine. Otherwise, we increase the contents of N by 1, and, unless we have reached the end of the list (and so are DONE), we jump back to COMPAR .

The entire sorting routine is

```
COMPAR:
C1:     MOVE    MEMWD(N)
        CAMLE   MEMWD+1(N)
        JRST    SWAP
        AOS     N
C2:     CAME    N,AC
        JRST    C1

DONE:   . . .

SWAP:   EXCH    MEMWD+1(N)
        MOVEM   MEMWD(N)
        JUMPE   N,C2
        SOJA    N,C1
```

In the COMPAR routine we have introduced AOS , the general format of which is

```
AOS         MEM
```

which adds 1 to the contents of MEM. (How could this be achieved using the ADDI instruction? Using the ADDM instruction? Write the respective routines.)

In the SWAP routine we have three new instructions. EXCH has the general format

```
EXCH        AC,MEM
```

and exchanges the contents of AC and MEM. Thus, the first two lines of SWAP interchange the contents of MEM(N) and MEM+1(N).

JUMPE is one of a group of commands; its format is

```
JUMPE       AC,LABEL
```

and its effect is to jump to the line bearing the given LABEL if the contents of AC are Equal to zero. Similarly, we have JUMPGE, JUMPL, JUMPLE, and JUMPN (jump if Not equal to zero). JUMPA Always jumps; JUMP *never* jumps, and so does nothing at all.

In the above program fragment, observe the destination of the JUMPE instruction! (Why is it to C2 and not to C1?) If N contains zero, we are back to the beginning of the list, and must move one place to the right. Otherwise, we move one place to the left and repeat the COMPAR routine. SOJA is Subtract One and Jump Always. The general format is

```
SOJA        AC,LABEL
```

it is equivalent to

```
SOS         AC
JRST        LABEL
```

The general form of the SOS instruction is

SOS MEM

which subtracts 1 from the contents of MEM.

In the program itself (Figure 2.5), there is an additional instruction SOJE AC,DONE at COMPAR. Notice that this instruction is carried out only once for each set of data, after all data has been read and before it is sorted. What would happen if we did not include this instruction?

Sorting problems occur so frequently that you should spend some time thinking out exactly how this routine works. You might like to practice using it to arrange randomly dealt playing cards in numerical order, or Scrabble letters in alphabetical order. Be careful to keep track of the contents of "AC" and "N" as you go. The flow chart in Figure 2.6 may be helpful here.

The complete program consists of a routine to read the numbers, and deposit them in a block of memory locations (we have allowed room for O 1000); then a routine like the above to rearrange them; finally a printout routine.

In the printout routine, some care has been taken over the format. The numbers are printed out in five columns. We achieve this by keeping a column count in accumulator CT. We start with 5 in CT. Every time we print out a number, we subtract 1 from the contents of CT. If the result is greater than 0, we print a TAB (ASCII code O 11). If the result is 0, we add 5 to start the process again, and print a ↵. We also print a carriage return and two line feeds after the last number, ready for input of the next collection of data.

Our READ routine allows all separators between numbers, except $ (ESCAPE), which it treats as terminating data input. To escape from the program, use ^C.

We need several accumulators for counting purposes, leaving scarcely enough for our printout

```
        AC=1
        N=2
        K=3
        CT=4
        INT=5
        DGT=6

START:  SETZM   AC
        SETZM   N
        MOVEI   CT,5
        SETZM   INT
        OUTCHR  [77]

READ:   INCHWL  DGT
        CAIGE   DGT,60
        JRST    SEP
        CAILE   DGT,71
        JRST    SEP
R1:     SUBI    DGT,60
        IMULI   INT,12
        ADD     INT,DGT
        JRST    READ

SEP:    MOVEM   INT,MEMWD(AC)
        AOS     AC
        SETZM   INT
S1:     CAIN    DGT,33
        JRST    COMPAR
        INCHWL  DGT
        CAIGE   DGT,60
        JRST    S1
        CAILE   DGT,71
        JRST    S1
        JRST    R1

SWAP:   EXCH    MEMWD+1(N)
        MOVEM   MEMWD(N)
        JUMPE   N,C2            ;if first item
        SOJA    N,C1

COMPAR: SOJE    AC,DONE
C1:     MOVE    MEMWD(N)
        CAMLE   MEMWD+1(N)
        JRST    SWAP
C2:     AOS     N
        CAME    N,AC            ;are we DONE?
        JRST    C1

DONE:   SETZM   N
        OUTCHR  [15]
        OUTCHR  [12]

PRINT:  MOVE    INT,MEMWD(N)
        SETZM   K
P1:     IDIVI   INT,12
        MOVEM   DGT,REMS(K)
        AOS     K
        JUMPN   INT,P1
P2:     SOJL    K,FORM
        MOVE    REMS(K)
        ADDI    60
        OUTCHR
        JRST    P2

FORM:   CAML    N,AC
        JRST    FINIS
        SOJG    CT,TAB
        OUTCHR  [15]
        OUTCHR  [12]
        MOVEI   CT,5
        SKIPA
TAB:    OUTCHR  [11]
        AOJA    N,PRINT

FINIS:  OUTCHR  [15]
        OUTCHR  [12]
        OUTCHR  [12]
        JRST    START

MEMWD:  BLOCK   1000
REMS:   BLOCK   20

        END     START
```

FIGURE 2.5 A program to list a collection of numbers in increasing numerical order.

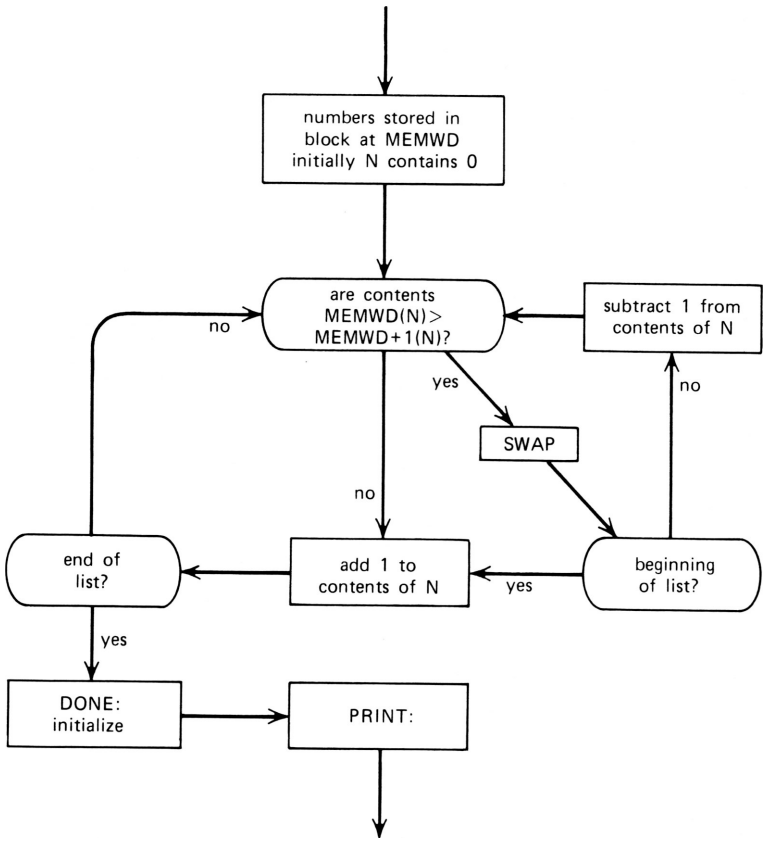


FIGURE 2.6 Flow chart to accompany the program in Figure 2.5.

routine. In more complicated programs, it would be very burdensome to have to reserve so many accumulators for printout. The solution is to string out the successive remainders in a block of memory words, using an accumulator for indexing. We have done this in our program, using accumulator K.

Notes:
We have used several new instructions.

SOJ- AC,LABEL

means: Subtract One from AC, and then, under the circumstances given by what follows J, Jump to LABEL.

SOJA	jump always
SOJE	jump if, *after* subtracting 1, (AC) = 0
SOJG	jump if, *after* subtracting 1, (AC) > 0

similarly with SOJGE, SOJL, SOJLE, SOJN.
Warning:

SOJ subtracts 1 and *never* jumps

it is equivalent to SOS AC .
The instructions

AOJ- AC,LABEL

Add One to the contents of AC, then compare (AC) with 0, and Jump accordingly.

The SKIP- instructions compare the contents of a memory location with 0, and skip a line accordingly.

```
SKIPE        MEM
```

skips if (MEM) = 0. SKIPA always skips.
Warning: SKIP alone *never* skips.
We also have SKIPG, SKIPGE, and so on.
We used SKIP- only in the routine FORM , to avoid an unwanted TAB after ↵.

Routine P1 strings out the remainders; routine P2 prints them out successively. In P1, the contents of K are increased by 1 after storing each remainder, in preparation for the next one. But this means that we enter P2 with K "pointing" to the location one beyond that at which the last remainder is stored. This is why, in P2, we decrease the contents of K by 1 before printing out the contents of the location to which K points. (How does this compare with our treatment of AC?)

In the complete program, we have placed routine SWAP before routine COMPAR, whereas in our earlier discussion the opposite was the case. Why does this make no difference?

Exercises: (i) Data is stored in locations starting at MEM. The number of locations is given by the contents of accumulator I. Write routines to . . .
(a) delete from storage all null items (i.e., when the location contains zero), by moving subsequent data down to replace them. Of course, the order of nonnull data items may not be changed, nor may their number be increased by both moving an item down and leaving it in its old location.
(b) set to zero all data found after any zero data item.

(ii) Check your routines by including each of them in a program that first places successive integers in a suitable block of locations starting at MEM. Work through the programs using DDT. Zero a few data items before running and check that every instruction serves its intended purpose.

(iii) Write a program to read decimal numbers typed in at the terminal, and store each number on input in increasing numerical sequence in a block of locations starting at MEM. Do this by going through the locations until the correct place for the new input is reached, then moving all subsequent numbers up one location to make room. Do not repeat any number already stored. When end of input is signaled, have your program print out the numbers in sequence, and start again.

Does your program still work if your second input contains fewer numbers than the first?

(iv) Write a program to read two numbers and print out their quotient to one hundred decimal places, properly rounded. (Hint: "teach" the computer grade school long division.) Show the decimal point in your output as a period (ASCII O 56).

*(v) Write a program to read two numbers and print out the period of the decimal expansion of their quotient (for example, 3/7 = 0.428571 with period 6).

2.4 WORD FORMAT

Recall that, as we stressed in Section 1.3, the computer understands only binary numbers. So what happens when we enter an instruction like MOVEM AC1,AC2? Well, it is easy enough to find out directly, using DDT; you should in any case be taking every opportunity gradually to familiarize yourself with the contents of Appendix A. Write a program, however trivial, with this instruction, setting AC1=1, AC2=2. If you check with DDT the contents of the word representing that instruction, you should find the value 202 040 000 002 . What on earth has this got to do with the original instruction? Certainly a great deal, for if you ask DDT to give you these contents as an instruction you will indeed get MOVEM AC1,AC2, possibly with machine numbered locations 1 and 2 replacing the names you gave to them. Clearly in some fashion the two are equivalent.

The equivalence is the outcome of a translation process, which is invoked when you execute a program with the extension .MAC. The translation is effected by a program known as an *assembler.* It is a very straightforward translation process for individual instructions: to each mnemonic instruction there corresponds a binary code, and vice versa. We could write our programs directly in the binary code, but doing so would require substantial and utterly pointless feats of memory. We can happily use mnemonic codes, and let the assembler do the rest; although we need to know what the assembler is doing.

Execute the following experimental program. Try to work out in advance what will happen.

```
        AC1=1
        AC2=2
        AC3=3
START:  MOVEI   AC1,130
        AOS     LAB
LAB:    MOVEM   AC1,AC2
        OUTCHR  AC3
        EXIT
        END     START
```

On the face of it, this program puts the ASCII code for X into AC1, moves it into AC2, and then inexplicably chooses to print out the contents of AC3! At the start of program execution, all memory locations normally have contents equal to zero. Yet this program prints out an X. How does an X get into AC3? Even better, if you amend the program to print out the contents of AC2, you will see that the X never reaches AC2!

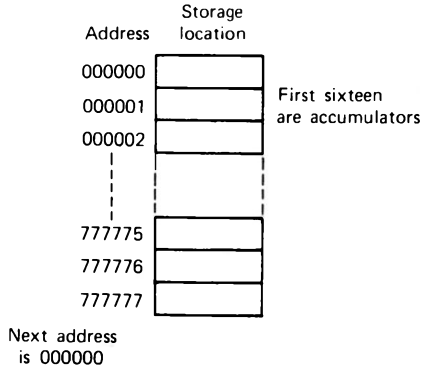
To find out what is going on, DEBug the program. Check after executing each instruction the contents of AC1, AC2, AC3 and the line labeled LAB, with the latter both as binary code and as an instruction. Assuming that AC2 and AC3 contain zero at the start, using DDT in octal constant type-out mode you should get

AC1	AC2	AC3	LAB
130	0	0	202040,,2 ← MOVEM AC1,AC2
130	0	0	202040,,3 ← MOVEM AC1,AC3
130	0	130	202040,,3 ← MOVEM AC1,AC3

for the effects of the first three instructions. When the program is assembled, the label LAB is everywhere replaced by the integer that is the address of the word representing the line bearing that label (DDT will tell you the address it uses). The contents of that address are equivalent to the instruction MOVEM AC1,AC2. The instruction AOS LAB adds 1 to the contents of that address, and as we have seen the result is the instruction MOVEM AC1,AC3. Since AC2=2 and AC3=3, we deduce that the 2 on the far right in the octal code for MOVEM AC1,AC2 refers to AC2. Indeed a much more general statement holds:

in any instruction of the form
instr AC,MEM
the address of MEM *occupies the right half of the binary code word generated.*

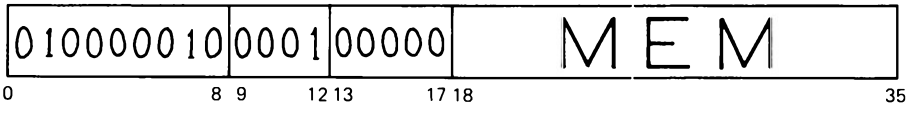
Thus, any memory location available to the user can be addressed using 18 bits; these locations must be numbered between 0 and O 777 777.



It is, however, not usually a good idea to use numbers rather than names for memory locations (except for accumulators). If you use names, the monitor will assign memory locations for them; with numbers, until you know just what you are doing, there is a chance of trying to access a location unavailable to you, with consequent program failure.

So the left half of the binary code word for an instruction must contain both the code for the mnemonic instruction, in this case MOVEM, and a reference to the accumulator, in this case AC1. The instruction code is contained in the leftmost nine bits. The bits are always numbered decimally 0 through 35, starting from the left; so the instruction code occupies bits 0 through 8. Nine binary digits correspond to three octal digits, and the code for MOVEM is O 202 = B 010 000 010. The process of associating the code with the actual operation to be carried out is a matter of hardware engineering, and need not concern us here.

Bits 9 through 12 of the instruction code word contain the accumulator number; this ranges from 0 through O 17, and so can occupy up to four bits. The bit pattern for MOVEM 1,MEM now looks like



To write the left half as an octal number, group the digits in threes

B 010 000 010 000 100 000 = O 202 040.
↑ - -code- - ↑ ↑ -AC- ↑

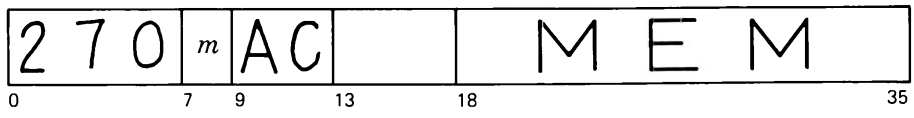
So the 4 refers to AC1! This confusion is unfortunate, but is an unavoidable consequence of the way in which grouping the binary digits in threes to obtain the octal equivalent cuts across the bits reserved for the accumulator reference.

Exercises:
(i) Use DDT to discover for yourself the three digit octal codes for the instructions IMUL, IMULI, IMULM, and IMULB.
(ii) What does IMULB do?
(iii) Using the notation X,,Y to specify a word whose left half (reading from the rightmost bit of the half word) contains X and whose right half contains Y, what is the instruction whose octal code is 202400,,4 ? What about 200400,,4 ?
(iv) Compare your results in exercises (i) and (iii). Can you draw any conclusions? Test them on ADD, SUB, and DIV.

Modes

Instructions referencing an accumulator and a memory location, and performing an arithmetic operation or a move are available in various forms. For example, integer multiplication is available as IMUL, IMULI, IMULM, and IMULB. These different forms are called *modes*. The mode, for such instructions, determines, as appropriate, which location is to be the *source* of data, and which is to be the *destination*. After your researches in the above exercises, you will not be surprised to learn that in all these instructions, bits 7 and 8 of the instruction code determine the mode.

The *basic* mode, like ADD, or MOVE, always has 0 in bits 7 and 8. It is concise to denote the format of an instruction like ADD, whose three digit octal code is 270, as



in which *m* denotes the mode. This can give the erroneous impression that somehow O 270 is to be squeezed into the seven bits numbered 0 through 6. In fact, the notation means that the O 270 is

to take up the nine bits, numbered 0 through 8, required to guarantee the housing of a three digit octal code; the presumption is that the code is such that bits 7 and 8 will be zero. Whatever represents the mode is then to be entered in bits 7 and 8.

Although we could regard the code for ADD as being two octal digits occupying bits 0 through 5 (which two octal digits?), this would lead to inconsistency with other arithmetical operations. For example, the code for SUB is O 274, which cannot be truncated to two octal digits; note that it still yields zeros in bits 7 and 8.

The contents of bits 7 and 8 determine the mode in the following way.

Bit 7	Bit 8	Mode
0	0	basic
0	1	immediate
1	0	to memory
1	1	to both

For MOVE instructions, "to both" is replaced by "to self," which is discussed later in this section.

Knowing that the code for SUB is O 274, the above table tells us that the code for SUBI is O 275; for SUBM it is O 276; and for SUBB it is O 277. If you are at all confused about what SUBB does, write a program to find out.

The fact that the memory reference occupies the right half of the instruction code word has a very important consequence for the Immediate mode instructions. Only the rightmost 18 bits of the data specified in an immediate mode instruction will be taken into account. If an immediate mode instruction is given with data item X, the assembler will take the rightmost 18 bits of X — let us call this |X — and form a word containing |X in its right half and zero in its left half: that is, 0,,|X . Thus, MOVEI AC,27 is equivalent to

```
        MOVE    AC,MEM
```

if before the END statement we have

```
MEM:    27
```

However,

```
        MOVEI   AC,1000000
```

will not work. Its effect will be the same as SETZM AC (why?). But we can achieve the desired result by

```
        MOVE    AC,MEM
```

with later on

```
MEM:    1000000
```

or

```
MEM:    1,,0
```

Another way uses direct representation of data, enclosed in *square brackets*:

```
        MOVE    AC,[1,,0]
```

Such a representation of data is called a *literal*. The assembler creates a table of all such literals, and puts into the instruction the appropriate address in the literal table. We have encountered literals before. (Where?)

Exercises: (i) Investigate the difference between MOVEI AC,−1 and MOVE AC,[−1].
(ii) How could the SUBI instruction be used in a sequence to set the contents of AC to −1 ?
(iii) Investigate the representation of negative numbers in a computer word. Use DDT,

with constant type-out mode, to display the contents of AC after each execution of the line labeled START in the following program

```
        AC=1
START:  SOJA    AC,START
        END     START
```

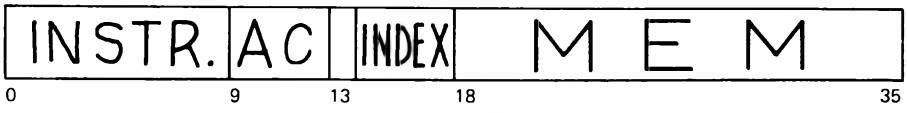
Begin with AC containing a small positive integer, and watch what happens. Can you make any sense of it? Does it perhaps remind you of an automobile mileage indicator, run backwards?

Effective Address

We have seen that in an instruction the code representing the operation itself occupies the nine bits 0 through 8 of the word. And whatever the instruction, any memory location to be referenced is contained in the right half of the word. If an accumulator is specified as such (rather than merely as a particular case of the memory reference), its number will be found in bits 9 through 12 of the word.

If an accumulator is used as an *index register,* its number will always be housed in the same place, for every assembler language code that references a memory location: this is in the four bits 14 through 17. Remember that any accumulator except accumulator 0 may serve as an index register.

Bit 13 has a special function, which we shall consider later. For the present, we shall assume it is set to 0. Thus we have the general instruction format



The control unit of the computer that goes through your program, performing your instructions line by line, is called the *central processor.* In the execution of any instruction, the first thing that the central processor does is calculate what is called the *effective address.* The procedure is

(i) retrieve the right half of the word;
(ii) if bits 14 through17 are set to zero, there is no indexing. Otherwise, add to the address determined in step (i) the contents of the accumulator whose number is given by bits 14 through 17.
(Plus a step (iii) if bit 13 is set to 1.)

The effective address calculation is carried out for every instruction whose format specifies a memory reference,* regardless of whether the result of the calculation will be used. For example, the instruction SKIPA will be understood by the central processor as SKIPA 0 . The effective address calculation will be carried out (and will yield zero) despite the fact that the effect of SKIPA MEM is wholly independent of MEM.

Since the effective address calculation is done before anything else when an instruction is performed,

> *there is no way at all in which an instruction can have any effect on its own effective address calculation.*

The effective address calculation is just the same for an instruction in immediate mode. The difference appears when the instruction itself is carried out. In other modes the effective address is

*With the exception of certain instructions used in running the system; these require special privileges and are not discussed in this book.

regarded as the address of the information on which the instruction will operate; that is, the required information forms the *contents* of the effective address. In immediate mode, however, *the effective address itself* is the required data item for the operation.

Negative Numbers

We end this section with a brief discussion of the representation of negative numbers in a computer word. Bit 0 of the word is called the *sign bit*, and is set to 1 for negative numbers. However, as you have seen in the exercises, the representation of −1 is far from being that of 1, save only for a 1 rather than a 0 in bit 0. The architecture of the DECsystem-10 uses the convention known as *twos complement* to represent negative numbers. If X is a positive number, to form the representation of −X

(i) form the representation of X; since X is positive, the sign bit will be set to 0, and bits 1 through 35 will contain the binary code for X;
(ii) subtract 1;
(iii) change all 0's to 1's and all 1's to 0's.

Familiarity will make this seem less mysterious, so pay careful attention to the exercises. Exercise (ii) will show you how to form the twos complement of an octal number directly.

Exercises:

(i) What is the octal code for
(a) MOVEI 1,WORD(3)
(b) IMULM 2,WORD(2)
(c) SUB 17,WORD
given that the assembler has assigned WORD to location O 6357.

(ii) What is the octal computer code representation for the octal negative numbers: −1703; −400 000 000 000; −1; −2; −10.

(iii) What is the largest octal number that can be represented in a single computer word? How is its negative represented? Is this the negative number of greatest magnitude that can be represented in a single computer word?

(iv) Does IMUL give correct results when both operands are negative?

(v) Find out what IDIV gives as quotient and as remainder when one or both operands is negative.

(vi) Write a routine to print out a decimal number comprising any number of digits (as long as a computer word will hold it) that will work for both positive and negative numbers. (Hint: the ASCII code for − is O 55.)

(vii) Whether the contents of a word are to be regarded as data or instruction is not something that the computer can "know" in advance; it depends on how the word is treated in your program. Suppose a text editing program is searching for the text /th through a series of locations starting at MEM and indexed by accumulator 12, with the comparison taking place in accumulator 15. Would the following sequence be the right sort of thing to do?

```
        MOVE    15,TEXT
TEXT:   ASCIZ   '/th'
        CAME    15,MEM(12)
```

(viii) What is the effect of an instruction SOJA 1,LAB(1) when accumulator 1 contains the number 1?

The negative of the contents of a word, in proper 36-bit twos complement form, can be formed by the MOVe Negative instruction MOVN. It is available in basic, Immediate and to Memory modes. So AC can be set to contain −1 by MOVNI AC,1.

In the same category as MOVE and MOVN we have the MOVe Magnitude instruction MOVM . MOVM AC,MEM has the same effect as MOVE AC,MEM if the contents of MEM are positive or zero; but if the contents of MEM are negative (that is, if bit 0 of MEM is set to 1), then it is equivalent to MOVN AC,MEM. The contents of the source word, here MEM, are unchanged.

The category is completed with the MOVe Swapped instruction MOVS, which interchanges the two halves of the source word, and puts the result in the destination word. The contents of the source word are unchanged.

All of the instructions MOVE, MOVM, MOVN, and MOVS are available with the mode suffices I, M, and S. I and M are by now familiar, and S means "to self." The self mode treats the memory location as both source and destination. It will also put the result in AC, as long as AC is not accumulator 0. If AC is accumulator 0, it is unaffected by an instruction in self mode.

Your answers to the following exercises should be checked using a suitable program. Be sure you understand what is meant by saying that a word contains, say, 1,,−1 . This is valid assembler language terminology, as in declaring an initial value

```
MEM:    1,,-1
```

or in a literal

```
        MOVE    AC,[1,,-1]
```

The double comma separates the two halves of the word. To contain 1, the left half of the word must have its lowest order (rightmost) bit set to 1, and its bits 0 through 17 set to 0. The right half of the word must contain the proper 18-bit twos complement representation of −1; that is, all its bits must be set to 1.

Exercises: Suppose that accumulator 0 contains 1,,−1; accumulator 1 contains −1; accumulator 2 contains −1,,1; accumulator 3 contains 17; and accumulator 17 contains 2.

(i) Which of the following instructions causes a skip?

(a) SKIPG
(b) SKIPG 2
(c) CAIN 1,−1
(d) CAML @(17)

(ii) What is the effect of each of the following instructions on the given accumulator contents?

(a) MOVSM 2,1
(b) MOVMS 2,1
(c) ADD (17)
(d) ADDI (17)
(e) MOVSS 3,17
(f) MOVSS 17,3
(g) IMUL 1
(h) IMUL 1,2
(i) MOVNS

CHAPTER THREE

PROGRAM STRUCTURE

3.1 SUBROUTINES

In the last chapter, we stressed the block structure of our programs. The general approach was to divide the problem we want to tackle into small sections. For each section we then write a routine, and the complete program is made up of the collection of routines together with various connections; such as jump instructions, between them. When the program is executed, it proceeds through the various routines in some order. Typically, the order has been: input, calculation, output.

More complicated problems, however, may lead to programs that *branch.* That is, there may be various routines leading from or to any given routine. For example, we might want to have results printed out at different stages of execution. We cannot readily do this using just one PRINT routine, because when printout is finished, the program has lost track of the point it had reached before jumping to PRINT ; so it cannot pick up the calculation at the point where it left off. The crux of the matter is that our PRINT routine is, once written, a fixed entity of the program. We can jump to it from as many points as we like; but there is only one way to leave it, that which is written into the routine.

It is true that we could terminate the PRINT routine with conditional jump instructions, and manipulate these to set us back to the point from which we jumped. But this would be very complex and cumbersome. It would also be pointless, since we have at our disposal the possibility of creating a *subroutine,* which is designed exactly with this difficulty in mind.

Before dealing with how to write subroutines let us consider their effects. If PRINT has been written as a subroutine, then printout is achieved by the Jump to SubRoutine instruction JSR

```
JSR        PRINT
```

and not by any of the jump instructions previously considered. The JSR instruction is said to *call* the subroutine. When PRINT has run its course, operations will continue automatically from the next instruction after the one that called PRINT .

Using subroutines, the approach to complex programming tasks is much simplified. In our initial sketch of a program, we would not trouble to consider the mechanics of, for example, a

printout routine. If at some stage we have a quantity in location MEM which we want printed out, we might write simply

```
PRINT    MEM
```

in our first draft, just as if PRINT were an assembler language instruction. Of course, it is not; so before running the program we would replace the above line by JSR PRINT , and write the appropriate subroutine.

This is a good approach whenever some task must be carried out many times. To begin with, we suppose that there is a single instruction to accomplish the task. Then, when the structure of the program has been worked out, it is time to fill in the necessary subroutines.

Now let us consider how to write a subroutine. Because we are used to writing the various parts of a program as separate routines, we can concentrate on the differences between routines and subroutines. Suppose we want to convert routine PRINT to subroutine PRINT . Then JRST PRINT must be replaced by JSR PRINT

Since a subroutine will return us to the mainstream of the program, a record must be kept of the location in the program at which the subroutine was called. We need not worry about how to do this. When the program is assembled, a memory word is formed for each instruction. Use of the JSR automatically stores the address of the location to which the subroutine will return us; this is the line after the JSR instruction. All we need to do is give the program room to store that address. The place for this is the first line of the subroutine. So instead of

```
PRINT:    first instruction
```

and so on, we have

```
PRINT:    0
          first instruction
```

The line bearing the label PRINT is now available for storing the return address.

```
PRINT    [return address]
```

Indirect Addressing

To leave the subroutine, we use the instruction

```
JRST     @PRINT
```

in which the symbol @ indicated *indirect addressing.* There must be no space between @ and PRINT. The symbol @ implies that the destination of JRST is not the location labeled PRINT, but rather the address stored at the location labeled PRINT.

Indirect addressing can be used with any memory location. Consider the instruction

```
MOVEI    AC,MEM
```

This puts in AC the *address* of MEM. The addresses of memory locations are just numbers. At assembly time, an address is assigned as MEM; thereafter, the expression MEM is considered identical with the number that gives that address. So the MOVE Immediate instruction moves that number into AC. Distinguish:

```
(a)      MOVE     AC,MEM
(b)      MOVEI    AC,MEM
```

(a) moves the *contents* of MEM into AC;
(b) moves the *address* of MEM into AC.

Suppose that (b) has been done. To move the contents of accumulator NUM into MEM, instead of

```
MOVEM    NUM,MEM
```

we can now use

```
MOVEM    NUM,@AC
```

a facility very useful in handling lists of data.

For example, suppose we have stored data in a block of memory words starting at MEM and ending at WRD, and that we want to carry out some check on the data items. As a simple example, we might want to replace any item less than D 100 by zero. A company not interested in outstanding accounts of less than one dollar might do exactly this (assuming that the number stored represents cents). We start by putting the addresses of MEM and WRD into accumulators INT and NUM. To save on move instructions, we hold the key quantity D 100 in the accumulator named HNDRD. Since D 100 = O 144, we could do this by MOVEI HUNDRD,144 , but we take this opportunity to indicate how to introduce a number as a decimal number. To do this, precede the number by ^D.

This is not CONTROL-D. *On this one occasion, we mean up-arrow* ^, *then* D.

The "main program" goes through the list, adding 1 to the contents of INT until the contents of NUM are reached. Since NUM contains the address of WRD, where the last data item is stored, there is no more to be done.

```
        MOVEI   INT,MEM
        MOVEI   NUM,WRD
        MOVEI   HNDRD,^D 100
LABEL:  CAMLE   INT,NUM
        JRST    FINIS
        JSR     CHECK
        AOJA    INT,LABEL
CHECK:  0
        CAMLE   HNDRD,@INT
        SETZM   @INT
        JRST    @CHECK
```

The CHECK subroutine replaces any item less than D 100 by zero.

In this simple fragment, using a subroutine takes up more instructions than not doing so. As we shall see, this is not always the case.

Note that the address of the first data item is lost. How could this be avoided?

Exercise: Expand the above fragment into a complete program. Use a subroutine to read in data items, and another to print them out.

A Text Editing Program

Now we shall use subroutines and indirect addressing to write a text editing program, which appears in Figure 3.1. The program is of a very limited nature, and the reader is invited to make improvements.

Our program will delete any superfluous spaces between words and after punctuation marks; allowing one space between words and after all punctuation marks except a period, after which it will allow at most two spaces. It will ignore one space at the start of a new line, but will treat two or more spaces as indications of a new paragraph.

Input text will finish with $ (ESCAPE); we shall put its ASCII code into accumulator ESC (INCHWL reacts to $, just as it does to ↵). Similarly, we use other accumulators to hold the ASCII codes of characters to which the program will make frequent reference.

Using accumulator LIST to hold addresses, the whole input routine is:

```
        MOVEI   ESC,33
        MOVEI   LIST,MEM
LAB1:   INCHWL  @LIST
        CAME    ESC,@LIST
        AOJA    LIST,LAB1
```

```
        TAB=1
        LF=2
        ESC=3
        SP=4
        COM=5
        PER=6
        COL=7
        SCOL=10
        LIST=11

START:  MOVEI   TAB,11
        MOVEI   LF,12
        MOVEI   ESC,33
        MOVEI   SP,40
        MOVEI   COM,54
        MOVEI   PER,56
        MOVEI   COL,72
        MOVEI   SCOL,73
LAB0:   MOVEI   LIST,MEM
        OUTSTR  MESSGE
LAB1:   INCHWL  @LIST
        CAME    ESC,@LIST
        AOJA    LIST,LAB1
        MOVEI   LIST,MEM        ;reset to beginning
        JSR     PARA
LAB2:   AOS     LIST
        CAMN    SP,@LIST
        JSR     SPACE
        CAMN    COM,@LIST
        JSR     PUNCT
        CAMN    SCOL,@LIST
        JSR     PUNCT
        CAMN    COL,@LIST
        JSR     PUNCT
        CAMN    PER,@LIST
        JSR     PERIOD
        CAMN    LF,@LIST
        JSR     PREPAR
        CAME    ESC,@LIST
        JRST    LAB2
        MOVEI   LIST,MEM
        OUTCHR  [15]
        OUTCHR  [12]
        OUTCHR  [12]
LAB3:   CAMN    ESC,@LIST
        JRST    LAB0
        OUTCHR  @LIST
        AOJA    LIST,LAB3

PARA:   0
        CAME    SP,@LIST
        JRST    @PARA
        AOS     LIST
        CAME    SP,@LIST
        JRST    P2
        SOS     LIST
        MOVEM   LF,@LIST
        AOS     LIST
        MOVEM   TAB,@LIST
P1:     AOS     LIST
        CAME    SP,@LIST
        JRST    @PARA
        SETZM   @LIST
        JRST    P1
P2:     SOS     LIST
        SETZM   @LIST
        AOS     LIST
        JRST    @PARA

SPACE:  0
S1:     AOS     LIST
        CAME    SP,@LIST
        JRST    @SPACE
        SETZM   @LIST
        JRST    S1

PUNCT:  0
        AOS     LIST
        JSR     SPACE
        JRST    @PUNCT

PERIOD: 0
        ADDI    LIST,2
        CAME    LF,@LIST
        JSR     SPACE
        JRST    @PERIOD

PREPAR: 0
        AOS     LIST
        JSR     PARA
        JRST    @PREPAR

MEM:    BLOCK   1000

MESSGE: ASCIZ   /

INSERT TEXT:
/

        END     START
```

FIGURE 3.1 A text editing program.

The AOJA instruction increases the contents of LIST by 1, and jumps back to LAB1. Now, therefore, @LIST refers to one memory location beyond that used on the previous occasion. In this way, characters are taken into successive locations.

The editing part of the program starts again at location MEM, and goes on until it encounters the $ character (which it suppresses in printout).

Upon encountering a space, we jump to subroutine SPACE which suppresses any further spaces. It does this by replacing the ASCII code for space with 0, which is the ASCII code for the "null character." But, with punctuation marks, we want to allow one space, or two in the case of a period, before suppressing further spaces. So, instead of jumping straight to SPACE, we jump to the respective preparatory subroutines PUNCT and PERIOD . These move on the required number of characters, then jump to subroutine SPACE to suppress any more spaces. We have here the phenomenon of a subroutine calling a subroutine—this is referred to as *nesting* of subroutines. Suppose we encounter a period. Then PERIOD moves on two characters, and calls SPACE. SPACE returns by the instruction JRST @SPACE to the line after JSR SPACE in PERIOD. But this line is JRST @PERIOD , which returns us to the mainstream of the program.

At the beginning of the text, we go directly to PARA to see if indenting is required. Later, we

check for a new paragraph on encountering a line feed. The preparatory subroutine PREPAR moves us on to the character after the line feed.

We have included in this program a message telling us to insert text, which will appear every time editing is finished and the program is ready for new input. The command to output a "string" of text is OUTSTR . What follows OUTSTR is the label of a line where the text string is to be found. In our program we have

```
        OUTSTR    MESSGE
```

so the text found at the line labeled MESSGE will be output.

At the line labeled MESSGE, we must first declare that we are going to give ASCII text. This is done by the declaration ASCIZ . After this comes at least one space or TAB; then the text, between *delimiters*. The delimiter is the first nonspacing character after the ASCIZ declaration; it must precede and terminate the text, but is not treated as text itself. Convenient delimiters are /, ' and ". The text may, as in our program, extend over several lines, until the original delimiter turns up again.

Exercises: (i) What do you suppose would result from:

```
        OUTSTR    MS
MS:     ASCIZ     'I'm all right.'
```

(ii) What is the purpose of the line

```
        CAME      LF,@LIST
```

in subroutine PERIOD ?

(iii) Fully annotate the text editing program in Figure 3.1 with your own comments, and draw a flow chart for it.

(iv) Try to amend the text editing program so that a single space after a period is expanded to two spaces.

Effective Address Calculation

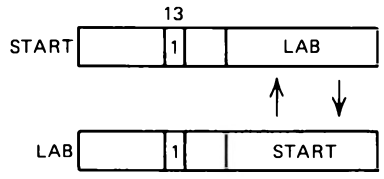
Prefixing the memory reference with the symbol @ causes the assembler to set bit 13 to 1 in the code word representing an instruction. Bit 13 is called the *indirect bit*.

In Section 2.4 we considered the effective address calculation carried out by the central processor at execution time on bits 14 through 35 of the instruction code word. The given memory reference is modified by indexing, if any, to get a new address. If bit 13 is set to 0, this completes the effective address calculation.

If, however, bit 13 has a 1, then the address found so far is not the effective address. Instead, the processor takes the contents of that address, and performs the whole effective address calculation over again on them! The process is exactly the same, with this new address as the starting point. So if the indirect bit in that address contains a 1, yet another level of address retrieval is demanded, and so on. The process continues until a word is retrieved with a 0 in bit 13. Then *its* right half, modified by the contents of the accumulator given by the contents (unless zero) of *its* bits 14 through 17, forms the effective address for the *original* instruction. Remember that the entire effective address calculation is completed before anything else is done in the performance of an instruction. Consider the perverse program

```
START:  JUMP    @LAB
LAB:    JRST    @START
        EXIT
        END     START
```

which will tie up computer time endlessly without result. You run this program at your own peril! The effective address calculation for the first instruction has the central processor retrieving the locations labeled START and LAB in eternal alternation.



The central processor never gets to the point of noticing that the instruction JUMP does nothing at all!

There are many instructions in which the effective address calculation is carried out, just as we have described, in spite of the fact that the information contained in the right half of the instruction code word is not regarded by the programmer as an address. One example is the Test instruction TRNE

```
        TRNE    AC,MASK
```

MASK here is a number (of at most 18 bits) provided by the programmer; if every bit in this number *that is set to 1* corresponds in position to a bit in the right half of AC *that is set to 0*, then the instruction causes a skip. The condition for a skip is that every bit *masked* by MASK must be set to 0. Consider for example

```
        TRNE    AC,1
```

The mask is 1, and so the only bit in AC to be tested is bit 35. So this instruction skips if, and only if, bit 35 of AC is set to 0. It tests whether the contents of AC are an even or an odd number.

```
        TRNE    AC,1
        JRST    ACODD
ACEVEN: ...
```

Similarly, the instruction TRNE AC,3 causes a skip precisely when the contents of AC are divisible by four. TRNE is mnemonic for Test the Right half of AC (with No modification of AC) and skip if Every masked bit equals 0.

The entire effective address calculation is performed for a TRNE instruction, although to the programmer the right half of the code word is merely a collection of bits acting as a mask. Suppose, for example, that accumulator 5 contains 1; then

```
        TRNE    AC,1(5)
```

is equivalent to

```
        TRNE    AC,2
```

and will cause a skip if bit 34 of AC is 0. If the instruction is

```
        TRNE    AC,@1
```

then the contents of accumulator 1 are retrieved as the starting point of a new effective address calculation. Suppose that accumulator 1 contains 23,,5 . Bits 13 through 17 of accumulator 1 are therefore set to 10 011; the indirect bit is set to 1, and the index register field of the word indicates indexing by the contents of accumulator 3. The memory reference retrieved from the right half of accumulator 1 is 5. Thus, the effective address calculation on the contents of accumulator 1 yields @5(3). For the next stage, the contents of accumulator 3 are added to the number 5. Since the indirect bit is 1 at this level also, the result is regarded as an address, and its contents begin the next stage of the effective address calculation. When the calculation eventually ends (by retrieving a word with indirect bit set to 0), the result is the mask for the original TRNE instruction.

It is possible to check the indirect bit in AC by

```
        TLNE    AC,20
```

This instruction tests the Left half of AC against the given mask; hence it will skip if the indirect bit is set to 0. (Why?)

Exercises: (i) How would you check whether
(a) the index register field (bits 14 through 17)
(b) the accumulator field (bits 9 through 12) of AC is set to zero?
*(ii) Write a routine to check whether the effective address calculation for the instruction at the line labeled LAB will yield an accumulator.

If the left half of AC contains zero, then (AC) and @AC give the same address. (Why does the *left* half of AC make any difference?) In such a case, it is always better to use indexing rather than indirect addressing because the former is much faster. Our text editing program is grossly at fault in this respect, but was, of course, designed as an illustration of indirect addressing.

It is impossible to substitute indexing for indirect addressing in the return from a JSR . For example, JRST @PRINT cannot be replaced by JRST (PRINT). PRINT is the name of a memory location that is not an accumulator; and only accumulators (other than accumulator 0) may serve as index registers. If you do this by mistake (try it!), on execution you will get a *Q-warning,* indicating that the assembler has found Questionable language. The assembler would then do its best for you; parentheses have a meaning for it beyond their use in indexing. If EXP is any 36-bit expression, then (EXP) indicates simply that its two 18-bit halves are to be interchanged. MEM(EXP) is the word formed by adding the result and the address MEM. In an example we ran, the instruction labeled PRINT was assembled into location 4724. JRST has the instruction code 254. Interchanging the two halves of the expression PRINT gives 4724,,0. This was added to 254000,,0 in the assembly of JRST (PRINT), giving 260724,,0 (octal addition!). This is indeed an instruction, of a type we shall learn later: its mnemonic code is PUSHJ 16,@0(4). Obviously, Q-warnings should be heeded!

Note that we have *not* described two different meanings for parentheses. For example, (7) is assembled as 7,,0; 4724(7) as 7,,4724. The assembler inserts the 7 into what the central processor will treat as the index field for an instruction.

Upon performing a JSR PRINT instruction, how does the central processor determine what to put into the location named PRINT (which you have left free)? It refers to one of its own internal registers, the 18-bit *program counter*, referred to as PC. At execution time, the central processor sets PC to contain the address given in the END statement of your program. In the line

```
END     START
```

START is the operand supplied by the programmer for the assembler language statement END . Now START is the label of some line of the program; and when the program is assembled, START is the name of the corresponding memory location. Hence the address of that location forms the initial contents of PC. The central processor now performs the following steps:

(i) retrieve the contents of the location addressed by PC;
(ii) increment the contents of PC by 1;
(iii) carry out the instruction given by the last word retrieved;
(iv) go to step (i).

In step (i) the central processor places the contents of the left half of the location addressed by PC into its 18-bit internal instruction register, and the address part into its 22-bit internal memory address register.

Observe that now step (ii) increases the contents of PC, so that when an instruction is being carried out PC contains the address of the next location after the instruction. We shall see exceptions to this later, when an instruction is *executed*—that is, performed out of the sequence given by PC. Note that in any case the sequence given by PC need not be consecutive, as many instructions (such as skips and jumps) change PC itself.

In step (iii) the central processor first finds the effective address, using for each level of retrieval the contents of the memory address register and the indirect bit. It then performs the instruction, which, together with any accumulator operand, is in the instruction register; this is done in special registers within the central processor. As observed above, the instruction may modify PC; for example, skip instructions add 1 to the contents of PC (why 1, rather than 2 ?).

Upon finishing with one instruction, the central processor begins again on the current contents of PC. To the central processor, the location whose address is in PC always contains an instruction. This is why the sequence in Exercise (vii) at the end of Section 2.4 goes wrong. The central processor has no way of passing over the location named TEXT as containing data rather than an instruction.

The internal registers of the central processor are not part of the memory available to a program. However, the only internal register with which the programmer is likely to be concerned is PC; and while it cannot be referenced directly, its contents can readily be moved into any memory location. For example, JSR PRINT places the contents of PC into the right half of the location named PRINT, and replaces the contents of PC with the address PRINT+1. Note that location PRINT contains the address of the location next after the JSR PRINT instruction, because PC was incremented before performing this instruction.

The effect of the instruction JRST is merely to replace the contents of PC with the address specified in the instruction. So JRST @PRINT is indeed the correct return from a subroutine.

Exercises: (i) If we wanted, under certain circumstances, to return to the second line after JSR PRINT, could we do it by JRST @PRINT+1? If not, how could we do it?

(ii) Using DDT, go through any program containing a JSR instruction, and see for yourself that the return address is put into the right half of the first word of the subroutine. What happens if you forget to leave the first word of the subroutine free?

Flags

If you have fully understood the process of effective address calculation, your findings in Exercise (ii) above will have caused you some concern. As you have seen, an instruction JSR PRINT not only puts the return address into the right half of the location named PRINT, but also stores something in the left half. What if bits 13 through 17 of the left half were not all zero, so that an attempt to return by JRST @PRINT could fail? (Why could it fail?)

The computer designers have, however, taken this into account. The half word stored in the left half of the address given in a JSR instruction always has its bits 13 through 17 set to zero, so there can be no further indexing or indirect addressing to upset the return procedure.

The left half actually contains information regarding the states of the various *flags*. The flags are indicators of certain circumstances arising in the course of the computer's operations. For example, you are already aware that the 36-bit limitation on the size of a word can cause arithmetical operations to produce incorrect results. If AC contains 200 000,,0, representing 2^{34}, then IMUL AC,AC produces zero; while the result of ADD AC,AC is 400 000,,0, representing -2^{35} (correct magnitude, but wrong sign). In each case, the operation has yielded more information than can be squeezed into a single word, and some has been lost. When such an event occurs, the fact is recorded within the central processor. The usual rather pleasing image is to think of the appropriate flag being raised; more prosaically, the appropriate bit is set to 1.

For our purposes, we can regard the various flags as being contained within an 18-bit register FLAGS, and say that a JSR moves the contents of FLAGS into the left half of the specified address. Relatively few of the bits in FLAGS will concern us (remember that bits 13 through 17 are always 0).

Exercise: Devise experimental programs to answer the following questions:

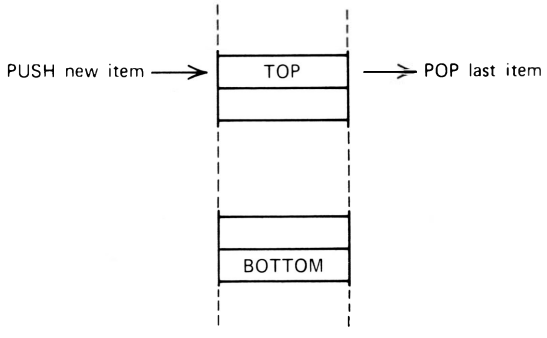
(a) do the two arithmetic overflows we discussed above set the same flags? (A flag is *set* if the bit representing it in FLAGS is set to 1; otherwise it is *clear*.)

(b) what happens if, when an overflow condition occurs, the appropriate flag has already been set by a previous overflow?

(c) what flags are set by an attempt to divide by zero? What actually happens if such an attempt is made?

3.2 PUSHDOWN LISTS

A *pushdown list* is a collection of items stored in such a way as to make the most recently stored item the most readily accessible. A common analogy is the type of plate holder found in a cafeteria. An extra plate is "pushed down" onto the top; in computer terminology, one acquires a plate by "popping" it up from the top. We shall use this terminology because it conforms with that of the MACRO-10 instructions, with a warning not to be misled into thinking that a whole list of stored items in computer memory is moved when a new item is pushed down or the latest addition is popped up. Pushdown lists are also called *stacks*, and in many ways the analogy is better: a new item is put on, or the last one taken off, the *top* of a stack. At the *bottom* of the stack is the first item put on, which will be the last to be taken off.



The subject of pushdown lists is by no means as obscure as might appear on first impressions. Successive quotients in a printout routine, and the start addresses of successive nested subroutines, are two of the many examples of information that is handled on just such a "last in, first out" basis.

Our first illustrative program, in Figure 3.2, mimics the collection policy of a (we hope) mythical utilities company. For each payment, the company keeps a record of whether payment was timely (enter T), later (enter L) or very late (enter V), by storing 0, 1, or 2. If payment is late on three, or very late on two successive occasions, supply is disconnected. Since only the most recent payments are of interest, a pushdown list is indicated.

Let us study the program of Figure 3.2, ignoring the first instruction for the moment. The instruction INCHRW requests the monitor to INput a CHaRacter; if no character has yet been typed, the monitor will Wait (and the program will not proceed) until one is. This is not a "wait on line" instruction; the monitor will react as soon as you press a key. Note that in this program, input of anything except T, L, or V leads to CUTOFF.

Suppose payment is timely. Then no previous payments need be checked. Accumulator STATUS is being used to hold the information to be pushed down onto the list. So we set STATUS to contain zero, and then

```
PUSH        P,STATUS
```

In the instruction PUSH AC,WRD the effective address holds the information to be pushed down onto the list. Indexing and indirect addressing on WRD are allowed in the usual way. Somehow, of course, the instruction has to find out where the list is. This information must be put into the accumulator AC referenced, before the PUSH instruction is used: AC must contain the *pushdown pointer*. And memory space for the list must be reserved.

In our program we have reserved, starting at MEM, enough space to record twelve payments. Accumulator P holds the pushdown pointer, and the first instruction of the program sets it up.

The IOWD statement causes the assembler to create a word in the special format required for a pushdown pointer. IOWD X,Y assembles as −X,,Y−1. So we have set P to contain −15,,MEM−1 (octal notation). Notice that the left half has been set to contain the negative of one more than the number of words the list may contain. The right half contains an address one less than that of the start of the list.

```
        STATUS=1
        P=2

START:  MOVE    P,[IOWD  15,MEM]  ;pushdown pointer
        OUTSTR  MESS1
        INCHRW                    ;waits for input
        CAIN    124
        JRST    TIMELY
        CAIN    114
        JRST    LATE
        CAIN    126
        JRST    VLATE

CUTOFF: OUTSTR  MESS2
        EXIT

TIMELY: SETZM   STATUS
        PUSH    P,STATUS          ;push down new status
        JRST    START+1

VLATE:  POP     P,STATUS          ;pop up last status
        CAIN    STATUS,2
        JRST    CUTOFF
        PUSH    P,STATUS          ;push down last status
        MOVEI   STATUS,2
        PUSH    P,STATUS          ;and new status
        JRST    START+1

LATE:   MOVE    STATUS,(P)        ;check last status
        JUMPE   STATUS,L1
        MOVE    STATUS,-1(P)      ;and last but one
        JUMPN   STATUS,CUTOFF
L1:     MOVEI   STATUS,1
        PUSH    P,STATUS          ;push down new status
        JRST    START+1

        0
        0
MEM:    BLOCK   14                ;0 14 = D 12

MESS1:  ASCIZ   /

PAYMENT PLEASE!   /

MESS2:  ASCIZ   /

SORRY, SUPPLY DISCONNECTED
/

        END     START
```

FIGURE 3.2 A program to simulate a mythical utilities company.

The instruction PUSH AC,WRD

(i) adds 1 to the contents of each half of AC;
(ii) moves the contents of WRD to the location now addressed by the right half of AC;
(iii) if the left half of AC contains zero, sets the appropriate flag.

So if initially P contains −15,,MEM−1, twelve successive PUSH instructions will deposit data into locations MEM through MEM+13 (octal). P will now contain −1,,MEM+13. A thirteenth PUSH will deposit data into MEM+14, which is not one of the locations reserved by us for the purpose. This does not, however, get any chance to cause problems. The left half of P now contains zero, and so the appropriate flag is set. This *not* one of the flags available to the ordinary user in FLAGS. The effect is to transfer control forthwith to the monitor, which will stop the program and print the message

?pdl ov at user PC *address*

Knowing the PC value at which pushdown list overflow occurred can be especially helpful when using DDT.

Execute the program in Figure 3.2, and respond to the request for payment with T, twelve times in succession. Now see for yourself the effect of one more T response.

If the response is V, so that payment is very late, the program must check the previous payment record. If a 2 was stored last time, we must go to CUTOFF. So we pop the last record up into accumulator STATUS with

```
POP     P,STATUS
```

The instruction POP AC,WRD

(i) moves the contents of the location addressed by the right half of AC into WRD;
(ii) subtracts 1 from the contents of each half of AC;
(iii) if the left half of AC contains −1, sets the appropriate flag.

POP sets the same flag as PUSH. POP will overflow if the left half of AC contains the 18-bit twos complement representation of −1; that is, if all the bits are set to 1. Since PUSH will overflow if the count in the left half of AC reaches zero, the count cannot be more than −1 when the first POP instruction is issued. This will reduce the count to no more than −2 before checking for overflow; so it is hard at present to see what use the overflow condition is for POP.

The point is that we can set up the pushdown pointer to cause overflow either by PUSH or by POP, but not by both. By starting the count negative in the left half of AC, overflow will occur if an attempt is made to store more information than available storage allows. This is what we have done in our program.

On the other hand, we can start the count at zero by setting up the pushdown pointer with MOVEI P,MEM−1. In this case PUSH will never cause overflow, because the first PUSH increases the count to 1 before checking to ensure that it is not zero. But now POP will cause overflow, as soon as an attempt is made to take out more information than has been put in.

Later we shall learn how to keep control, rather than let it pass to the monitor, when pushdown list overflow occurs. Until then, it does not make much difference which way we set up the pointer because the program must be written so that overflow never occurs. For consistency, we shall continue to use a negative starting count.

Having popped up the last payment, if it too was very late (a 2 was stored), we then go to CUTOFF. Otherwise, we push the previous record down again, then push down a record of the current payment. We return to START+1 to demand the next payment. (Why not return to START ?)

If payment is late, we again pass to the appropriate routine. In this we illustrate a way of referencing items in a pushdown list without popping them up. Since nothing is removed from the list in this routine, the new payment record is stored with just one PUSH. The address of the last item stored is in the right half of the pushdown pointer; if this is not clear to you, look again at our description of the action of the PUSH instruction. So we can retrieve the item into STATUS by

```
MOVE    STATUS,(P)
```

We repeat: this does not remove the item from the list.

In fact POP does not actually delete any item from its location in memory. But, by amending the pushdown pointer, POP makes the list look shorter from the point of view of the program. The location referenced by POP is no longer considered to be part of the list, and we speak of the item as being removed from the list.

Later in this routine we must refer to the last item but one in the list. We do this by indexing the mythical address −1 by the contents of accumulator P. The effective address calculation yields one less than the contents of the right half of P, which is just where the item we want is stored.

Exercises: (i) If you paid late on the last two occasions, and are unable to pay on time this time, your situation is still not hopeless. Find out why, and amend the program so that it does what the company management clearly intended.

(ii) How would you reference the *n*th previous payment, where *n* is the contents of accumulator AC ?
(iii) Could the first line of routine LATE be changed to MOVE STATUS,@P ?
(iv) The following routine is an attempt to use a pushdown list for storing the successive quotients in a print out routine. Does it work? Try it in a program of your own, and explain what happens.

```
        INT=1
        DGT=2
        P=3
        ...
        MOVE    P,[IOWD  13,MEM]
        ...
L1:     IDIVI   INT,12
        PUSH    P,DGT
        JUMPN   INT,L1
L2:     POP     P,DGT
        ADDI    DGT,60
        OUTCHR  DGT
        JRST    L2
        ...
MEM:    BLOCK   12
```

Application to Subroutines

The ideas of this and the last sections are very nicely combined in a subroutine calling instruction that stores the return address in a pushdown list. This, and the corresponding instruction that effects the return, are rather subtle and require careful attention, but they are useful enough to amply repay that attention.

The instruction

PUSHJ AC,LABEL

referencing a pushdown pointer contained in accumulator AC, and the line bearing the given LABEL,

(i) adds 1 to the contents of each half of AC;
(ii) if the left half of AC contains zero, sets the appropriate flag;
(iii) puts the contents of PC in the right half and the contents of FLAGS in the left half of the location *now* addressed by the right half of AC;
(iv) moves into PC the address of the location named LABEL.

Note that step (iii) stores in the pushdown list the address of the location next following the PUSHJ instruction. (Why?) So the correct return address is stored. Step (iv) is effectively a jump to LABEL.

```
        INT=1
        DGT=2
        P=3
        ...
        MOVE    P,[IOWD  27,MEM]
        ...
        PUSHJ   P,PRINT
;returns here after PRINT routine
        ...
PRINT:  IDIVI   INT,12
        PUSH    P,DGT
        SKIPE   INT
        PUSHJ   P,PRINT
        POP     P,DGT
        ADDI    DGT,60
        OUTCHR  DGT
        POPJ    P,
        ...
MEM:    BLOCK   26
```

FIGURE 3.3 A printout routine.

In Figure 3.3 we have made a better job of the printout routine we tried to write in Exercise (iv) above. The routine PRINT is called by PUSHJ P,PRINT , where P has been set up to contain the pushdown pointer. After each remainder has been put on the list by a PUSH instruction, we return to PRINT to find the next remainder, again using a PUSHJ. This continues until we have found all the digits for printout.

For example, suppose we start with accumlator INT containing D 3154. When the instruction SKIPE INT finally causes a skip, the pushdown list will look like this:

MEM:

flags	address 1
	4
flags	address 2
	5
flags	address 2
	1
flags	address 2
	3

Address 1 is the location following the first PUSHJ instruction; note that this is the address to which control should return after completing the print out routine. Address 2 is the location following the other PUSHJ instruction; this is the address of the instruction POP P,DGT.

Exercise: Work out why the pushdown list has the appearance illustrated above. Use DDT to check.

After the instruction SKIPE INT has caused a skip, the digit 3 gets popped up off the list and printed out. Now we want to return to address 2, and pop up the next digit. The instruction

```
POPJ    AC,
```

(i) subtracts 1 from the contents of each half of AC;
(ii) if the left half of AC contains −1, sets the appropriate flag;
(iii) moves into PC the contents of the location that *was* addressed by the right half of AC before step (i).

Observe that the comma is needed after the reference to AC. The assembler interprets an address not followed by a comma as a memory reference. So if the comma were forgotten, the assembler, finding no accumulator reference, would assume that accumulator 0 was meant and assemble POPJ 0,AC . In fact POPJ requires no memory reference, and should have none.

So now the POPJ P, instruction pops "address 2" up off the list, and causes a jump to address 2; this is the POP P,DGT instruction. Next the digit 1 gets popped up and printed out. The same thing happens successively with the digits 5 and 4. The only item then remaining on the list is "address 1," and this is the location to which the POPJ P, instruction returns, finally leaving the printout routine.

There is no problem in using just one pushdown list to hold both data and jump addresses. You do have to be careful not to mix up the two kinds of information. It is not generally very useful to reference a jump address with a POP; and popping up data with a POPJ may be disastrous. (Why?)

Exercises: (i) What is the largest number that can be printed out by the routine in Figure 3.3?
(ii) Why is the first location of a subroutine called by a PUSHJ instruction not a null word, as it is when the subroutine is called by a JSR ?

Pushdown Pointer as Counter

Very often it is desired to have numerical output of a computer program in tabular form. Columns in such a table are normally right justified; that is, with all least significant digits in line with one another, as for example:

3154	8
72	1023
999	50

The correct number of spaces must be printed out before the leading digit, and the left half of the pushdown pointer provides a convenient counter for this. We have done this in the program of Figure 3.4. We must not amend the pushdown pointer itself, since it will be needed later; so we begin the spacing routine by copying the pushdown pointer into accumulator T. The left half of accumulator T now gives us a count of the number of spaces needed. We increment T after each space, using the instruction

AOBJN AC,LABEL

which Adds One to Both halves of AC, and Jumps to LABEL if AC is not Negative.

Observe that the condition for a jump with AOBJN is on the sign of the whole word AC. There will be a jump if and only if bit 0 of AC contains a 1, after the incrementation has been effected. In effect, the left half of AC is being checked.

```
        AC=1
        N=2
        P=3
        CT=4
        T=5

START:  MOVE    P,[IOWD  10,MEM]  ;pushdown pointer
        SETZB   CT                ;initialize
        AOS
        MOVEM   AC
        PUSHJ   P,S1
        JRST    .-3

S1:     IDIVI   AC,10             ;octal print out
        HRLM    N,(P)
        JUMPE   AC,.+3
        PUSHJ   P,S1
        SKIPA
        PUSHJ   P,S2              ;to format routine
        HLRZ    N,(P)
        ADDI    N,60
        OUTCHR  N
        POPJ    P,

S2:     SOJG    CT,.+4            ;column count
        OUTCHR  [15]
        OUTCHR  [12]
        MOVEI   CT,10
        MOVE    T,P               ;spacing routine
        OUTCHR  [40]
        AOBJN   T,.-1
        POPJ    P,

MEM:    BLOCK   10

        END     START
```

FIGURE 3.4 A program to print out numbers right justified in columns.

In our program, the AOBJN instruction jumps back one line to output another space, until the count in the left half of T reaches zero. Thereupon POPJ P, leaves the spacing routine. To jump back one line, we use the symbol . (a period), which is assembler language terminology for the address of the current line. Equivalently, the symbol . is equal to the contents of PC, less 1 (why "less 1"?). Earlier in the program we jumped to the third previous line by JRST .−3 . Similarly, an alternative to SKIPA is JRST .+2 ; in fact the latter is a faster instruction, as SKIPA has to reference memory. This is a convenient notation, and obviates endless labels. A danger is that one might amend a program, but forget to make the corresponding emendations to the jump instructions. For example, if an extra instruction is put into a routine, and the instruction AOJN AC,.−5 has been used to jump back in order to repeat the routine, then this instruction must be changed to AOJN AC,.−6 . Of course, the count is octal, although a jump of O 10 lines or more should certainly be handled with a label. There are enough ways of introducing bugs into programs, without adding the danger of miscounting lines to their number!

Because we are making no use of the flags stored by the PUSHJ instructions in the left halves of locations in the pushdown list, we can store data in those half words instead. Our program stores each remainder in the left half of the word stored by the last previous PUSHJ instruction. So if D 3154 were being printed out, when the jump to S2 took effect the pushdown list would look like this:

MEM:

4	address 1
5	address 2
1	address 2
3	address 2
flags	address 3

Half Word Instructions

To store our data in this way we need instructions for moving the contents of half words. The basic mnemonics for these are HRR, HRL, HLR, and HLL. The first letter stands for Half. The second letter specifies which half of the *source* word is to be moved, and the third letter specifies which half of the *destination* word is to receive it.

In the basic mode, the source is MEM and the destination is AC. For example

```
HLR     AC,MEM
```

moves the Left half of MEM into the Right half of AC. MEM is unchanged, and so is the left half of AC.

A suffix, however, may be used to indicate that the other half of the destination word should be amended: Z means set it all to Zeros; O means set it all to Ones. So

```
HLRZ    AC,MEM
```

has the effect of HLR AC,MEM and in addition sets the left half of AC to zero.

There is another suffix, E, standing for Extend. It places the leftmost bit of the source half word into all bits of the other half of the destination. This gives an easy way of changing the representation of a number from half word to whole word format. Suppose we have a number in the left half of MEM, and we want to use the whole word AC to house it. If the number is positive, then HLRZ AC,MEM is all that is required. But if the number is negative, stored in the left half of MEM in 18-bit twos complement form, then we would need HLRO AC,MEM (why?). However,

HLRE AC,MEM works whether the contents of the left half of MEM are positive or negative, since it has the effect of HLRZ in the first case, but of HLRO in the second.

After the suffix, if any, a final letter may be used to indicate the mode: Immediate, to Memory, or to Self. Thus

```
HRLM    AC,MEM
```

moves the Right half of AC into the Left half of MEM, leaving AC and the right half of MEM unchanged.

Note carefully that it is *not* the case that the second letter of the half word instruction mnemonics refers to AC, and the third letter refers to MEM. This is a mistake often made by beginners. In fact, the second letter refers to the source word, while the third letter refers to the destination word. Which of these is AC and which is MEM will depend on the mode.

A number in the right half of MEM may be extended to the whole of MEM by

```
HRRES   MEM
```

As always with the Self mode,

```
HRRES   AC,MEM
```

will also put the resulting contents of MEM (the full word result) into AC, as long as AC is not accumulator 0.

In Immediate mode, the memory reference is regarded as a word whose right half is the given number, and whose left half is zero. Thus HLLZI AC,X is equivalent to SETZM AC (why?).

Exercises: (i) In the program of Figure 3.4
- (a) where are "address 1" "address 2" and "address 3"?
- (b) how many spaces between columns will the program give?
- (c) what is the largest number that the printout routine can handle without causing pushdown list overflow?
- (d) what is the largest number that the printout routine can handle without upsetting the tabulation?

(ii) Write a routine to check whether AC and the right half of MEM contain the same number. If the number is negative, it is in 36-bit twos complement form in AC, 18-bit twos complement form in the right half of MEM. Remember that the left half of MEM might not be empty. You may alter the contents of AC if you wish, but not those of MEM.

(iii) Which instruction will cause a skip precisely when the right half of AC contains zero?

(iv) How would you change the contents of the right half of AC to
- (a) the negative of the original contents;
- (b) the magnitude of the original contents;

without affecting the other half? (Negatives to be in 18-bit twos complement form.)

*(v) Write a program to read and evaluate any arithmetical expression composed of integers, the minus sign, and parentheses. Parentheses here are meant only to specify the order of evaluation, not to indicate multiplication. (Hint: base your program on a subroutine EVAL that evaluates an expression from left to right until it encounters a parenthesis. Make EVAL respond to a right parenthesis with POPJ P, and to a left parenthesis with PUSHJ P,EVAL .)

Extras

We have seen several instructions that (together perhaps with other effects) add 1 to, or subtract 1 from, each half of AC. In addition, there is AOBJP which jumps if the contents of AC, after each

half has been incremented by 1, are Positive. How all these instructions are carried out depends on which model of central processor is installed. The older model KA10 processor forms the quantity 1,,1 and adds it to or subtracts it from the whole word. So there can be a "carry" from the right half into the left half. For example, if AC contained −1,,−1 (all 1's), then AOBJP AC,LABEL would leave AC containing 1,,0 . Check this for yourself, observing that a 1 carried out of bit 0 of AC is lost. (Which flag in FLAGS does this set?)

The more recent KI10 and KL10 central processors amend the two halves of AC independently, so there can be no "carry" from the right half into the left. Thus, if AC is set to all 1's, these instructions will lose a 1 carried out of each half of AC; this will leave AC containing zero.

Exercise: Write a program that will tell you whether or not your installation uses a KA10 processor. Notice that AC can be set to contain all 1's by any of MOVNI AC,1 , HRROI AC,−1 or HRREI AC,−1.

The introduction of a new model of central processor is generally accompanied by a few new MACRO-10 instructions. These will use operation codes that were previously unassigned. So if an instruction available only on the KL10 (the most recently introduced processor) is used with a KI10 or KA10, the monitor will stop the program and print an error message. The result will be similar if an instruction introduced with the KI10 processor is used on a KA10.

An instruction to ADJust the Stack Pointer is available on the KL10 processor only. The instruction

```
ADJSP   AC,X
```

will add the quantity X (which may be positive or negative) to each half of AC; the result is formed in AC.

This is a particularly useful instruction when various collections of data are contained in blocks in the same pushdown list. For example, suppose we want to move the results of successive calculations from location WRD into every fifth location in a pushdown list. This is accomplished if, after every time we

```
PUSH    P,WRD
```

we then

```
ADJSP   P,4
```

remembering the PUSH itself moves the pushdown pointer up one place. If each time we do this we also SUBI N,5 then accumulator N (if initially it contained zero) will record the total adjustment of the pointer. We can set it back to where we started by

```
ADJSP   P,(N)
```

If a positive adjustment in an ADJSP instruction changes the count in the left half of the pushdown pointer from negative to positive, then pushdown list overflow occurs. This is also the case if a negative adjustment changes the count from positive to negative. This ensures that the overflow checking facility built into PUSH / PUSHJ in the first case, or POP / POPJ in the second, is maintained.

3.3 PROGRAM CONTROL

When a program has to be written to perform a large and complex task, the first approach should determine only the general plan of attack. Subsidiary problems are left until later. For example, a complicated file sorting job might involve frequent interchanging of blocks of data. In the initial sketch it might be convenient to write

```
SWAP    K,MEM,WRD
```

to indicate that a number (given by the contents of accumulator K) of locations starting at MEM should have their contents exchanged with those of the corresponding locations starting at WRD.

Later, the subroutines must be written in detail. It is quite likely that this will produce ideas as to how the main program can be improved. Writing a large program often involves many stages in which the scope of a subroutine is slightly extended to enable simplifications to be made in the routines that precede its calls, and vice versa. It is also worth considering how best to write the subroutines so that they can be carried unchanged into future programs. For both present and future use, it is necessary to know

(i) what storage the subroutine uses;
(ii) what data must be passed to the subroutine;
(iii) where the subroutine delivers its results.

(i) Although this problem can be neglected when the first sketches are made, it can cause disaster if not properly dealt with later. Consider our SWAP "command" above. The subroutine that replaces it needs an accumulator to effect the word exchanges; a second accumulator is also called for as a counter if the contents of K will be needed later. It is of course essential not to alter accumulators containing necessary information. This can cause problems, as there might not be enough accumulators for each of many subroutines to have exclusive use of those it needs. A reasonable solution is to keep a block of memory locations at, say, TEMP, and assign them as needed to subroutines for temporary storage of accumulator contents. Thus, if TEMP+6 were the last such location so far assigned, and the next subroutine to be written required accumulators AC and N, it could begin with

```
MOVEM    AC,TEMP+7
MOVEM    N,TEMP+10
```

and end with

```
MOVE     AC,TEMP+7
MOVE     N,TEMP+10
```

before the return. Since each subroutine has its own temporary storage, no problem arises if one subroutine calls another. Of course, all this merely wastes time if other accumulators are unused.

The comments on each subroutine should include a list of the accumulators whose contents may be changed by the subroutine.

(ii) For example, our SWAP subroutine must know where locations MEM and WRD are, and which accumulator holds the number of words. This could be done quite simply by always using a particular accumulator for the word count, and by putting the addresses of MEM and WRD into the two halves of another specified accumulator, before calling the subroutine. On the other hand, the subroutine might be more useful if it could do its job regardless of where the data references were stored. If this is to be the case, the information needed by the subroutine must be passed to it as *parameters*. It may be necessary to define precisely a *calling sequence* for the subroutine, to show just how the parameters must be passed.

Item (iii) is really incorporated here. Any results should be returned to locations either fixed (and listed in the comments) in advance, or specified in the calling sequence.

Subroutine Jump Instructions

The program fragment in Figure 3.5 illustrates the process of passing parameters to a subroutine. To avoid unnecessary complications our subroutine is quite trivial; it merely calculates the unrounded average of a collection of numbers. Two parameters must be passed to the subroutine: the location of the first data item, and the accumulator whose contents are equal to the number of data items. Now

```
;data to be processed here contained in locations
;starting at MEM. Number of data items given by
;contents of CT. Calling sequence for routine AV is
        JSP     T,AV
        MEM             ;starting location
        CT              ;address of number of items
        ...             ;return to here
        ...
;averaging routine, no round up.  AC usage: K, INT
;result returned in INT
AV:     SETZM   INT
        MOVE    K,(T)
        ADD     K,@1(T)
        ADD     INT,-1(K)
        SOS     K
        CAMLE   K,(T)
        JRST    .-3
        IDIV    INT,@1(T)
        JRST    2(T)    ;return after data refs
```

FIGURE 3.5 Routine to illustrate passing parameters with JSP.

we could do this by

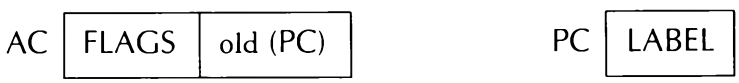
```
JSR         AV
MEM
CT
```

but it would be complicated. We would leave the first line of subroutine AV free for the address of the line after the JSR ; but in this case, that is not the correct return address. (Why not?) We would have to amend the address stored at location AV before returning, and this would be rather tedious.

It is much easier to use the subroutine calling instruction JSP that Jumps and Saves PC in an accumulator. The instruction

```
JSP         AC,LABEL
```

(i) places FLAGS in the left half of AC, and the contents of PC in the right half of AC;
(ii) moves into PC the address of the location named LABEL.



The disadvantage of JSP is that it takes up an accumulator to store the flags and the address of the location following the JSP instruction. The corresponding advantage is that one can return either by JRST @AC, or by JRST (AC) . The latter is not only faster, but can be amended to allow for locations taken up by parameters listed after the JSP instruction. In Figure 3.5 the parameters occupy two words; so the correct return is JRST 2(T) .

Exercises: (i) Why did we not leave a free line at AV ?
(ii) Study Figure 3.5 carefully, with particular regard to the various effective address calculations.
(iii) Amend the program so that the average is returned in the same accumulator that originally held the number of data items.

Where subroutines are nested (that is, where one subroutine calls another) the JSP instruction is not very suitable because each level of nesting requires a further accumulator. We end the list of subroutine calling instructions with one that, with its special return instruction, is particularly

```
;calling sequence for data swap routine
        JSA     T,SEARCH
        MEM             ;start of source file
        WRD             ;start of destination file
        ...             ;return to here
        ...
SEARCH: 0
        MOVE    I,(T)
        CAMN    (I)
        JRST    S1
        AOS     I
        SKIPE   (I)
        JRST    .-4
        JRA     T,2(T)          ;not in record
S1:     MOVE    N,1(I)          ;delete routine
        MOVEM   N,(I)
        AOS     I
        SKIPE   (I)
        JRST    S1
        JSA     T,INSERT
        JRST    S1-1            ;job done
        ...
INSERT: 0
        MOVE    I,INSERT        ;destination file
        MOVE    I,1(I)
        SKIPN   (I)
        JRST    .+3
        CAMLE   (I)
        AOJA    I,.-3
        EXCH    (I)
        AOS     I
        SKIPE   (I)
        JRST    .-3
        EXCH    (I)
        JRA     T,(T)           ;insertion done
```

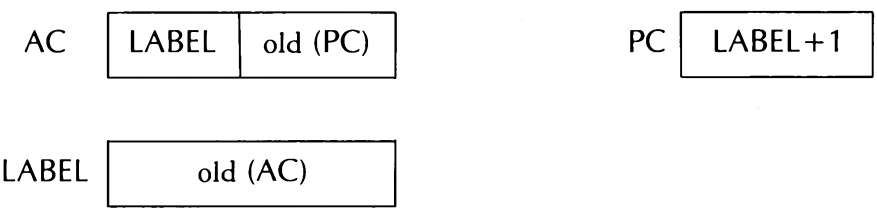
FIGURE 3.6 Routine to illustrate nesting subroutines with JSA—JRA.

suitable for passing parameters to nested subroutines. Our illustration, in Figure 3.6, is a routine to search a file for a given data item; and, if the item is found, to delete it from that file and insert it, in order, in a second file. The locations of the first words of the files must be passed as parameters.

The instruction Jump and Save Accumulator

JSA AC,LABEL

(i) moves the contents of AC to location LABEL;
(ii) moves the address the location LABEL to the left half of AC, and the contents of PC to the right half of AC;
(iii) moves into PC the address of location LABEL, plus 1.

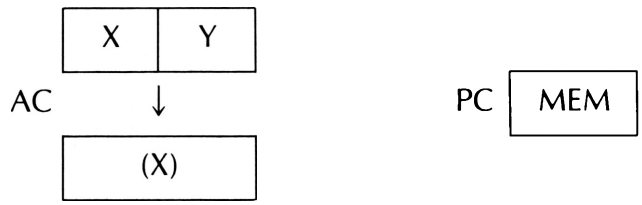


So although JSA requires an accumulator, it saves its contents in the first location of the subroutine called; this location must therefore be left free, as with a JSR.

The return from a subroutine called by a JSA instruction is effected by the Jump and Restore Accumulator instruction:

JRA AC,MEM

(i) moves the contents of the location addressed by the left half of AC into AC;
(ii) moves into PC the address of MEM.



The first step restores the original contents of AC. For the second step to return to the line following the calling JSA instruction, it is necessary that the address of MEM be equal to the contents of the right half of AC. That is, the memory location referenced in the returning JRA instruction must be (AC). Note that AC has already been amended by the first part of the JRA instruction at this stage.

Thus the correct return to the line after

```
JSA     AC,LABEL
```

is

```
JRA     AC,(AC)
```

If parameters are being passed, the correct return address is obtained by indexing the number of locations occupied by the parameters with index register AC. This is just the same idea as with a JSP instruction.

Exercises: (i) In the program fragment of Figure 3.6
- (a) what is the accumulator usage of subroutine SEARCH and its subroutine INSERT?
- (b) what test is used to determine when the end of a file is reached?
- (c) what happens to the location used to present the data item in question to the SEARCH and INSERT routines:
  - (1) if the item is found?
  - (2) if the item is not found?
- (d) what does line INSERT+2 do, and why?
- (e) consider the bottom line of the fragment; what are the contents of T, SEARCH, and INSERT both before and after this instruction is carried out? Which instruction is carried out next?

(ii) Write a calling sequence for subroutine INSERT so that it can be used independently of subroutine SEARCH.

Observe that we can make the location to which a subroutine returns depend on conditions detected by the subroutine. A subroutine to find the largest factor of a number might be called by

```
JSP     T,FACTOR
P                       ;holds number
JRST    PRIME           ;return here if prime
...                     ;here if not
```

If FACTOR puts the largest factor into accumulator AC, the return could be

```
CAIN    AC,1
JRST    1(T)
JRST    2(T)
```

The JSP instruction is very useful for examining the flags. We can simply read the flags into accumulator T by

```
JSP     T,.+1
```

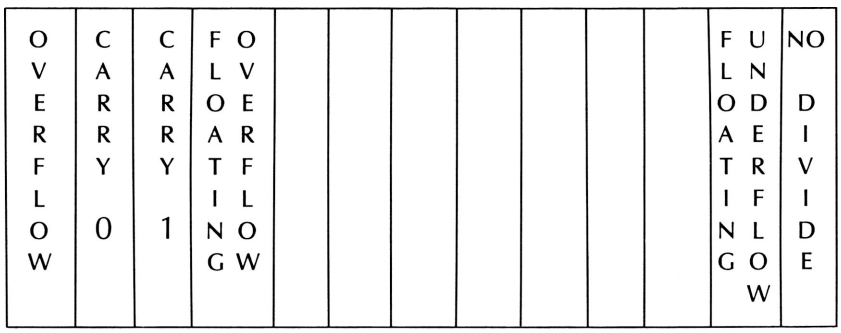


FIGURE 3.7 FLAGS

which then continues with the next instruction in sequence. Only the flags specifically marked in Figure 3.7 will be of interest to us.

Overflow

In complex arithmetical instructions it is essential to be cognizant of the states of the relevant flags. Overflow can occur within a calculation even when the correct result could readily be held within a single word. For example,

$$\frac{30.29.28. \ldots .16}{15.14.13. \ldots .1}$$

is a fraction in which either numerator or denominator alone would cause overflow; yet the fraction itself is well within bounds. But once overflow has occurred, it is not likely that the results will be correct if nothing is done about it. See this for yourself by writing a program to successively double the contents of AC by means of

```
(a)     ADD     AC,AC
(b)     IMULI   AC,2
```

Each of these will, if repeated often enough, produce nonsensical results because of overflow. Furthermore, each will produce different nonsense!

An ADD instruction can overflow in two different ways: two positive summands can produce too large a result; or two negative summands can produce a result of too large a magnitude.

Suppose we try to add $2^{34} + 1$ to itself. This number has a 0 in bit 0 since it is positive, a 1 in bits 1 and 35, and zeros elsewhere. The addition is performed like this

$$\begin{array}{r} 010\ldots001 \\ +\ 010\ldots001 \\ \hline 100\ldots010 \end{array}$$

which looks just right for binary addition, until we recall that bit 0 is the sign bit! The bottom line represents not the correct result $2^{35} + 2$, but $-2^{35} + 2$. Note that the result is *not* the negative of the correct result. But if we regard the bottom line as a 36-bit positive number, then it is the correct result. Similarly, the result when the addition of two negative numbers causes overflow is correct if regarded as a 36-bit negative number. In each case we are supposing that we have all but the sign bit of a 37-bit number.

There are many possible responses to the discovery of such an overflow; some of them are:

(i) stop the program; this is the normal response in a higher level language;
(ii) let the number take up another word; allowing as many words as may be needed to house all the binary digits of a number is the concept of multiple precision arithmetic;

(iii) lose accuracy by keeping only the most significant decimal digits of the result. This may be done by dividing the result by ten, and carrying on with the calculation. Of course care must be taken that this does not invalidate later calculations. It may be necessary to keep in a separate location a count of the number of times that ten has been divided out. If we need to add X and Y, but to avoid overflow X/10 has been stored, then the best thing is to form X/10 + Y/10, equal to (X + Y)/10. Then at printout, the count of divisions by (decimal) 10 indicates the number of terminating zeros required.

It is not quite straightforward to divide by ten after overflow has occurred, since we want to regard the sign bit as part of the number, and division instructions will not do this. We could divide by two if we could merely shift all the digits one place to the right, losing the rightmost one, and then adjust the sign bit. This is done by the Logical SHift instruction

```
LSH     AC,X
```

which shifts the contents of AC the number of bits given by the magnitude of the quantity X: to the left if X is positive, to the right if X is negative. Anything moved out of AC is lost; bits in AC vacated by the shift are set to zero. An effective address calculation is carried out for LSH; so a shift by the number of bits specified by the contents of accumulator CH is achieved by LSH AC,(CH) .

Here we shift one place to the right by

```
LSH     AC,-1
```

If overflow was caused by addition of positive numbers, then all we now need to achieve division by ten is

```
IDIVI   AC,5
```

Otherwise, the sign bit of AC must first be set to 1, since the sign bit of our mythical 37-bit number is properly carried in by a shift only when that bit is zero, for a positive number.

Let us now consider how in practice to deal in this way with the possibility that ADD AC,MEM might cause overflow. Overflow will always set OVERFLOW, which is bit 0 in FLAGS. Positive overflow will in addition set CARRY 1, which is bit 2 of FLAGS; negative overflow will instead set CARRY 0, which is bit 1 of FLAGS.

OVERFLOW is generally only set by instructions that have genuinely overflowed, and is indeed set under such circumstances by all arithmetical instructions. Some of these will also set either CARRY 0 or CARRY 1, *but not both.* Confusingly, however, CARRY 0 and CARRY 1 can be set together by a variety of innocuous conditions which cause no overflow; in such cases therefore OVERFLOW is not set. Remember also that

once any of these flags are set, they remain set until the program clears them.

Thus, there is no point in checking the flags after ADD AC,MEM unless we have cleared them first. This is done by a group of instructions that Jump if Flags are set, and CLear them. The instruction

```
JFCL    F,LABEL
```

assembles with the quantity F in the accumulator field, so F must represent a number between 0 and O 17. However, F does not specify an accumulator; rather, by setting certain of bits 9 through 12 of the instruction code word, it determines which flags should be examined. These four bits in the accumulator field correspond to the first four bits in FLAGS. Any combination of these flags may be selected by the appropriate choice of F. If one or more of the flags specified by F is set, the instruction will clear them, and jump to the address determined by the effective address calculation; if none of these flags are set, there is no jump. The table in Figure 3.8 lists all possible combinations, as well as six special mnemonics allowed for certain combinations.

JFCL LABEL (equivalent to JFCL 0,LABEL) is an example of a no-op—an instruction that does nothing at all. We shall see later that even a no-op can be useful; and since JFCL with zero

F	OVERFLOW	CARRY 0	CARRY 1	FLOATING OVERFLOW	MNEMONIC
0					JFCL
1				X	JFOV
2			X		JCRY1
3			X	X	
4		X			JCRY0
5		X		X	
6		X	X		JCRY
7		X	X	X	
10	X				JOV
11	X			X	
12	X		X		
13	X		X	X	
14	X	X			
15	X	X		X	
16	X	X	X		
17	X	X	X	X	

FIGURE 3.8 Flag selection in JFCL instructions.

accumulator field does not actually fetch the flags, it is the fastest no-op available. (What other no-ops have we encountered?)

We can clear OVERFLOW, CARRY 0 and CARRY 1 before our ADD instruction by

```
JFCL    16,.+1
```

which continues in sequence regardless of whether any of these flags were actually set. Afterwards we check OVERFLOW with the JOV instruction. We want to carry on if it is clear, but if it is set we must do other things first. The overflows may be handled in a subroutine OVRFLW; but JOV alone cannot be used to call the subroutine, as there would be no way to enable the return. Some trickery like this is needed:

```
JOV     .+2
SKIPA
JSR     OVRFLW
```

although there are faster skips than SKIPA. (For example?)

Exercises: (i) Write a program that will add together the contents of locations starting at MEM (the number of locations is housed in AC). The result should be in the form: *A* multiplied by 10 to the power *B*; where the numbers *A* (to as many significant figures as possible) and *B* are housed in accumulators.

(ii) Write a routine to print out a number stored in the form given by Exercise (i).

(iii) Check for yourself that SUB overflows when the magnitude of the result is too large, in just the same way as ADD.

A multiplication instruction that overflows sets OVERFLOW, but neither of the CARRY flags. However, the only thing generally worth doing if an IMUL instruction overflows is to stop the program. The product of an X-digit number with a Y-digit number may contain up to X + Y digits (regardless of the base). The IMUL instruction stores only the least significant 35 bits of the product, together with the correct sign; so more information may have been lost than mere awareness that overflow has occurred could recover. For example, the squares of 8×10^9 and 9×10^8 are both stored as zero.

The instruction IDIV sets OVERFLOW if the divisor is zero. It does so also if the dividend is -2^{35} and the divisor is 1 or -1. In any of these cases, no attempt is made to carry out the instruction, and the flag NO DIVIDE is set. This flag is not examined by any JFCL instruction, so to check it we must read the flags into an accumulator. Since NO DIVIDE is bit 12 in FLAGS, the sequence

```
JSP     T,.+1
TLNE    T,40
```

will skip if NO DIVIDE is clear. So a call to a suitable error subroutine can follow this sequence. To skip if NO DIVIDE is set, the second instruction should be

```
TLNN    T,40
```

This is Test the Left half of AC (with No modification of AC) and skip if Not every masked bit equals 0. (It is enough for a skip that even one masked bit is set to 1.)

If it is possible to continue the program after NO DIVIDE, this flag should first be cleared. Since NO DIVIDE is not set by any other kind of condition, it is all right to clear it after rather than before each check. First we set up the left half of accumulator T to contain the flags as we want them to be. If all flags are to be cleared, HLLI T, will serve. Otherwise we can clear the appropriate bit in T at the same time as we check it. With NO DIVIDE, for example, this is done by

```
TLZE    T,40
```

which Tests the Left half of T against the given mask, skips if Every masked bit is 0, and Zeros all masked bits in any case. Or the routine could be

```
JSP     T,.+1
TLZN    T,40
JRST    OK
```

followed by a routine for dealing with the NO DIVIDE condition. If it is possible to continue, the routine could return with JRST (T) since the TLZN instruction cannot now skip. (Why not?) But this does not reset the flags in FLAGS. We need instead the Jump and ReSTore the Flags instruction

```
JRSTF   (T)
```

Apart from the flags, this instruction has the same effect as JRST. Indeed, they are in the collection JRST X,LABEL where X indicates a particular function. For JRST alone, X is zero, while JRSTF is a mnemonic for JRST 2, . The other instructions in this collection are illegal for the ordinary user.

The instruction JRSTF will set the contents of FLAGS equal to the contents of the left half of the *final word retrieved* in the effective address calculation carried out for the instruction. This is subject to the proviso that no change will be made in certain bits of FLAGS that the ordinary user is not permitted to change. In the effective address calculation for the instruction JRSTF (T) the last word retrieved is T; so the left half of T is used to reset the flags, which is as we wanted.

Exercises:

(i) Could we restore the flags and jump to LABEL by JRSTF LABEL ? (Hint: which is the final address retrieved?)

(ii) Could we avoid, in the above sequence, repeating the instruction TLZN T,40 by returning with JRSTF 1(T) ?

(iii) The name *logical* is applied to instructions that regard a word merely as a collection of 36 bits; LSH is in this category. Besides LSH there is the Arithmetic SHift instruction ASH. The difference is that ASH regards a word as a sign bit plus 35 bits representing the magnitude of a number. Consequently, ASH never affects the sign bit; the shift works on bits 1 through 35 only.

Investigate the differences among LSH AC,1 ASH AC,1 and IMULI AC,2 for positive and negative contents of AC. Do the same with LSH AC,−1 ASH AC,−1 and IDIVI AC,2.

(iv) Determine the value of X if the sequence

```
ADDI    AC,X
ASH     AC,-5
```

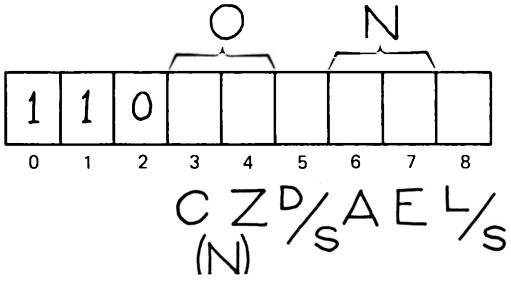
gives the same quotient in AC as IDIVI AC,40 when AC contains a negative integer.

(v) Can LSH or ASH set flags?

*(vi) Use your conclusions in Exercises (iv) and (v) to write a routine to replace division when the divisor is a power of 2.

Test Instructions

The format of the instruction code part of the Test instructions is



Since bits 0 and 1 are set to 1, and bit 2 to 0, the 9-bit instruction code for Test instructions is always between O 600 and O 677. All sixty four combinations give valid instructions. The mnemonics are three or four letter codes, of which the first is T. The second letter is R (right), L (left), D (direct) or S (swapped).

If the second letter is L or R, then on assembly bit 5 is set to zero. Also, bit 8 is set to 0 for R, to 1 for L. In these instructions, the effective address calculation itself yields the mask, which is compared with the appropriate half of AC.

On the other hand, D as the second letter assembles with 1 in bit 5, and 0 in bit 8. In this case, the contents of the effective address act as a mask for the whole word AC. S as the second letter indicates that the contents of the effective address with its two halves swapped will be the mask for the whole word AC; it assembles with 1 in bits 5 and 8.

Bits 3 and 4 are assembled according to the third letter, and indicate the modification prescribed for AC. If the third letter is N, both bits are 0, and there is No modification. If Z, bit 3 is 0 and bit 4 is 1; all masked bits are set to Zero. If C, bit 3 is 1 and bit 4 is 0; all masked bits are Complemented: 1's are changed to 0's and vice versa. The letter O assembles with bits 3 and 4 set to 1, and all masked bits are set to One.

The fourth letter, if any, indicates the conditions for a skip, and corresponds to bits 6 and 7. If there is no fourth letter, the instruction assembles with 0 in bits 6 and 7; these instructions never cause a skip. Letter A, however, corresponds to 1 in bit 6 and 0 in bit 7, and Always skips. Letter E assembles with 0 in bit 6 and 1 in bit 7, and skips if Every masked bit is 0. Finally, letter N assembles 1 in bits 6 and 7, and causes a skip if Not every masked bit is set to 0.

The possible combinations may be illustrated by

$$T\begin{Bmatrix}R\\L\\D\\S\end{Bmatrix}\begin{Bmatrix}N\\Z\\C\\O\end{Bmatrix}\begin{Bmatrix}-\\E\\A\\N\end{Bmatrix}$$

Exercises: (i) Suppose there were no MOVN or MOVM instructions. Write routines to simulate each.

(ii) Write a routine that tests the contents of a location to determine whether the location contains a Test instruction that always causes a skip.

(iii) Write a routine to determine whether a location contains an instruction that, conditional on a comparison, could jump more than O 200 locations (in either direction).

(iv) Which single Test instruction will increase all positive even numbers by 1, without affecting positive odd numbers? What will it do to negative numbers?

*(v) If AC contains zero, the instruction Jump if First Find One

```
JFFO        AC,LABEL
```

will set AC+1 to zero and continue in sequence. Otherwise it will count the number of zeros to the left of the first 1 in AC, place the count in AC+1, and jump to the instruction at LABEL. Write a routine using this with Test and shift instructions to check whether a positive number in AC is

(a) a power of two;

(b) a power of eight.

3.4 EXTENDED LANGUAGE CAPABILITIES

In this section we discuss various ways to extend the capabilities of MACRO-10 according to individual requirements. Let us begin with the problems of inputting and outputting data, which have occupied so much of our time in previous sections. We now have both of these processes systematized very satisfactorily. But do we have to write out the routines in full in every program in which we need to read and write data? That would be tedious enough, and would increase the danger of making typing errors.

A simple-minded way of avoiding this is to use the COPY command to the monitor. This command causes the monitor to run a special system program for manipulating files, called PIP, acronym for Peripheral Interchange Program. We could keep a copy of a print out subroutine in a file called PRINT. We would then make our file called TEST.MAC up to the point where we wanted to include the printout subroutine; then, after exiting from TECO, issue to the monitor the command

```
COPY TEST.MAC = TEST.MAC, PRINT ↵
```

which concatenates the *input files* (those named on the right of the = sign) into a single *output file* bearing the name given on the left of the = sign; this may be a new name, thereby creating a new file. There can be any number of input files. Then we would return to TECO and finish writing our file.

A similar procedure could be used to insert, at various points in the file, code that occurs frequently.

You might also like to familiarize yourself with the more advanced features of TECO (Appendix B) that enable all the above actions to be performed without returning to the monitor.

Macros

Our first approach to rather more sophisticated ways of dealing with these problems will enable us to create our own instructions, and use them very much like assembler language instructions. In this section we shall, for example, create the instruction IREAD, designed to read an integer typed in at the terminal. In our programs we shall be able to issue the instruction

```
IREAD       MEM
```

to read a decimal integer into MEM. To do this, we must of course have written the appropriate routine somewhere, in the appropriate format. What we write is called a *macro,* which is a facility for defining a single statement as representing a whole sequence of instructions.

As a simple example, suppose we want to interchange contents of individual memory locations frequently, and get tired of always having to write the three instructions, such as

```
MOVE    MEM
EXCH    WRD
MOVEM   MEM
```

needed to accomplish this. So we write a macro SWAP, such that our one instruction

```
SWAP    MEM,WRD
```

will suffice. The macro may be anywhere within the program that employs it. It is often convenient to put macros before the first instruction of the program itself.

The first line in a macro must be the special assembler language DEFINE statement. In this statement, the name that the macro is to bear follows as an operand; then come the *arguments* of the macro, in *parentheses.* In our example, the name of the macro is to be SWAP. The macro will require two arguments: the names of the locations whose contents are to be interchanged. In the macro, "dummy" symbols can be used for the arguments; that is, they do not have to be the symbols used when the macro is called. Single letters are convenient. So we might begin the definition of this macro with

```
DEFINE      SWAP    (X,Y)
```

In the DEFINE statement, the arguments should be enclosed in parentheses, separated by commas, with no intervening spaces or tabs within the parentheses.

The sequence of instructions that the macro name and arguments are to represent now follows. The locations referred to in these instructions bear the "dummy" names given to them in the list of arguments in the DEFINE statement. In this case, the two locations must be referred to as X and Y. The whole sequence of instructions to comprise the macro must be enclosed within *angle brackets* <...> . A complete defining sequence for this macro is

```
DEFINE      SWAP    (X,Y)
<           MOVE    X
            EXCH    Y
            MOVEM   X       >
```

In a program containing this macro, the SWAP "command" can now be used. Spaces and tabs are acceptable in the line calling the macro. Optionally, the arguments may be enclosed in parentheses. The operands in the line calling the macro replace the dummy arguments in the code generated by the macro, in the appropriate order. Thus, if the instruction is SWAP MEM,WRD then MEM replaces X and WRD replaces Y.

A comment before the DEFINE statement should indicate the purpose of the macro, and state its calling sequence and AC usage.

Exercises: (i) Write a trivial program using the SWAP macro. Using DDT, try to find out what actually happens when a macro is called.

(ii) Suppose we want to interchange the contents of MEM and WRD only if the contents of MEM are nonzero. Does the sequence

```
SKIPE   MEM
SWAP    MEM,WRD
```

do what is wanted?

The assembler will replace all macro calls that it encounters with the entire code of the macro itself. Once the program has been assembled there is no trace in the code that a macro has been used.

You can see this for yourself, even without using DDT. Suppose you gave the program you wrote for Exercise (i) above the name SWAP.MAC. To examine how SWAP is assembled, we must digress to look more closely at what goes on when we execute a program. The command EX causes the monitor to run a system program called COMPIL. This program acts as an intermediary between the user and other system programs that translate the user's programs into machine code, and prepare them for execution. Actual execution is accomplished when the monitor passes control to the user program.

The Loader

The process of preparing your programs for execution is accomplished by the system program known as the Linking Loader; the DECsystem-10 Linking Loader is called LINK-10. It is possible to pass commands directly to LINK-10; but for most purposes a collection of monitor commands that, like EX, cause COMPIL to run various features of LINK-10 will be adequate.

The first thing to do with a program you have written is to *compile* it. This produces a file with the same name, but the extension is now .REL instead of .MAC . This file consists of the translation of the program into binary code.

The command COMPILE causes the appropriate language translator to be called upon. Note that commands to the monitor can always be abbreviated, as long as the abbreviation is distinguishable from any other command; the first three letters are usually sufficient. The extension to the source file determines which translator is used. Programs in assembler language should be translated by the MACRO assembler, and this is implied by the extension .MAC . The extension need not be specified in the COMPILE command; so you could command

COM SWAP ↵

If the most recently created version of the source program has not already been compiled, a brief message will notify you that compilation has taken place. At present, however, you have no way of examining the contents of the file SWAP.REL now listed in your directory. The TYPE command is unhelpful. (Why?) You must instead include in the COMPILE command the subsidiary command, or *switch,* that will cause a *listing* of the binary file to be made. The switch for this purpose is designated by /LIST and the command should be

COMPILE SWAP /LIST ↵

There need be no space after the name of the file and before the switch; we have included one only for clarity.

If you have already compiled the most recent version of SWAP, and have not deleted SWAP.REL from your directory, there will be no further compilation (and hence no listing produced) unless you demand it by using the /COMPILE switch. In this case the command would be

COMPILE SWAP /LIST /COMPILE ↵

The order of the switches is unimportant.

The /LIST switch will add to your directory a file bearing the same name as the source file, but with the extension .LST. Now TYPE SWAP.LST ↵ will produce interesting information. (In some systems different names are created for such files. Check your directory in case of any difficulty.)

Let us examine the .LST file for a very simple program. In Figure 3.9 you can see the type out of TEST.LST. The source file TEST.MAC appears on the right in its entirety. On the left of each instruction is the code it generates.

The column to the far left of the source programs contains the address of each word in your

```
.MAIN   MACRO %52(551) 19:43 21-MAY-77 PAGE 1
TEST    MAC     21-MAY-77 19:38

                        000001                  AC=1
000000' 201 01 0 00 000060      START:  MOVEI   AC,60
000001' 051 01 0 00 000007'             OUTCHR  MEM
000002' 051 01 0 00 000001              OUTCHR  AC
000003' 350 00 0 00 000001              AOS     AC
000004' 307 01 0 00 000071              CAIG    AC,71
000005' 254 00 0 00 000001'             JRST    START+1
000006' 047 00 0 00 000012              EXIT
000007' 000000  000040          MEM:    40
                000000'                 END     START

NO ERRORS DETECTED

PROGRAM BREAK IS 000010
CPU TIME USED 00:00.115

5K CORE USED
```

```
.MAIN   MACRO %52(551) 19:43 21-MAY-77 PAGE S-1
TEST    MAC     21-MAY-77 19:38         SYMBOL TABLE

AC              000001
EXIT    047000  000012
MEM             000007'
OUTCHR  051040  000000
START           000000'
```

FIGURE 3.9 A program listing.

assembled program; these are numbered from zero up, with six digits allowed for each address. Then follows the code generated. Notice that in the code for instructions the listing separates the 9-bit operation code, the accumulator field, the indirect bit, the index field, and the memory reference. Data, however, is presented in half word format. Several other modes of representation of word contents are used; you will see this with listings of later programs. The format most relevant to the context of the word, and hence most helpful to the user, is always used.

Below this is the table of symbols defined by the user, together with their values. During assembly, MACRO searches for each symbol it encounters, first in the table of any user defined macros; then in its table of assembler language operation codes and monitor calls; then in the table of any user defined symbols (like AC, START, and MEM here); finally it searches a table of mnemonics for certain monitor calls, among which OUTCHR and EXIT are included, and lists any as if the user had defined them.

If the definition of a symbol is nowhere to be found within these tables, the listing will indicate this with the symbol * following the code generated, and will print error messages if appropriate.

Exercises: (i) Remove the line defining MEM from the program of Figure 3.9. Now obtain a listing of the program, and study the differences.

(ii) Choose one of the simpler programs from the earlier sections, and write out a listing as you think it would be; now compare your version with a listing obtained from the machine.

Note that statements such as AC=1 and END generate no code in the binary file. AC is entered with its value in the symbol table, and the position of the END statement is reflected in the *program break* information.

Compare the code for the lines AOS AC and JRST START+1. In each case, the address part is equal to 1: AC is accumulator 1 and START+1 is assembled as location 1. But the symbol ' next

to the 1 representing the address of START+1 is very important. It indicates that the code is *relocatable*; when it is loaded into core ready for execution, LINK-10 will decide where to put it, in terms of actual machine locations. Thus relocatable addresses should be regarded as being relative to some starting address, which itself is determined at load time.

Exercises: (i) Study Figure 3.9 with careful attention to understanding why some right halves of instruction code words are relocatable and others are absolute (not relocatable).
(ii) Get a listing for your program SWAP.MAC and study it carefully. What does the symbol ^ indicate?
(iii) What do you suppose .REL stands for?

The COMPILE command produces a file of relocatable binary code on disk. The next stage is to *load* the file into core. This is accomplished by the LOAD command to the monitor. If no .REL file is found, LOAD will cause one to be compiled from the appropriate .MAC file. Thus, if you are going to load immediately, there is no point in compiling first as a separate operation. The /LIST and /COMPILE switches may be used with LOAD. You might also try the /MAP switch, which creates a "map" of the loading process; see it by then typing the appropriate .MAP file. However, note that the TYPE command runs PIP in your core, thus destroying the version there of any program you have loaded (the *core image*). Of course the disk files are unaffected.

Files are loaded into core starting at location O 140; we shall see the significance of this later. Obviously all users cannot be loading their programs into the same locations; nor for that matter can all their references to accumulator 1 grapple for the same location number 1 in the machine. In fact, an absolute address is fixed only with regard to its position within whatever block of core the monitor may make available to the user (the user's *virtual address space*). However, the process of mapping the memory addresses assigned by you within your virtual address space into physical memory within the machine need not concern you at all. You can regard your own absolute addresses as if they were physically fixed. Every user can think of her or his accumulator 0 as if it were the only location number 0 in the computer.

Now load a file, and examine the contents of location O 140 by issuing to the monitor the E (Examine) command

E 140 ↵

and observe that the first instruction of your program is there. Any location available to you may be examined with an E command. The specified address must be octal, and the contents will be typed out as two octal half words.

To execute a loaded program, just enter the command

START ↵

which will start execution from the starting address specified in your program. Assuming that this is the first instruction, and your program has been loaded into locations beginning with location O 140, then

START 141 ↵

will start execution from the second line of your program, and so on.

If the program TEST will be run more than once, it is a good idea to avoid having to repeat the loading process every time, by *saving the loaded version* with

SAVE TEST ↵

The saved file will have the extension .SAV . This is a binary file in a format that is both rapidly retrievable for execution and compact for economical disk storage. Note that it may only be formed from the *loaded* version of a file. Once the .SAV file has been created, the .REL file will normally be of no further use, and should be deleted from disk storage.

On future occasions, TEST can now be executed by

RUN TEST ↵

or brought back into core by

GET TEST ↵

and then, when the "job set up" message appears,

START ↵

Now we are ready to return to the subject of macros.

Assembly of Macros

It is very important to be aware that a macro call is assembled by *direct substitution* of the actual code defined by the macro. Since only rarely will a macro generating just one line of code be written (although we give an example of one below), SKIP will not skip forward over a macro, nor will JRST .−2 skip back over it. Labels must be provided much more generously in programs that use macros.

For the same reason, long macros can give rise to long programs, which thus take longer (and cost more) to assemble and load. In contrast, a subroutine is assembled only once in a program. Later we shall see how to combine the advantages of both methods.

In Figure 3.10 we have the promised macro IREAD. Consider first only the first three arguments. A call

```
IREAD       MEM,2,10
```

within a program will cause two octal integers to be read into locations MEM and MEM+1. (Why octal?) In the line CAIL 16,60+B note the facility for having calculations performed *by the assembler.* Compile a program containing calls to this macro with different values of B, and see for yourself that 60+B is already evaluated before the program is run. The assembler will evaluate expressions involving user defined symbols, numbers, and arithmetical operations +, −, * (multiply), / (divide

```
;MACRO to read N (default 1) integers, base B
;(default decimal), into locations starting
;at X (default AC 1).  Call by IREAD  X, N, B
;AC usage:  15, 16, 17

DEFINE  IREAD (X<1>,N<1>,B<12>,%L1,%L2,%L3)
<       SETZM   17
%L1:    CAIL    17,N            ;all read?
        JRST    %L3
        SETZM   15
        INCHWL  16
        CAIG    16,40       ;superfluous separators
        JRST    .-2
%L2:    CAIGE   16,60           ;accept only digits
        JRST    ERR             ;in number system
        CAIL    16,60+B         ;of specified base
        JRST    ERR
        SUBI    16,60
        IMULI   15,B
        JOV     ERR
        ADD     15,16
        INCHWL  16
        CAILE   16,40
        JRST    %L2
        MOVEM   15,X(17)
        AOJA    17,%L1
%L3:    >
```

FIGURE 3.10 Macro to read a number.

and discard any remainder); pairs of *angle brackets* <...> may be used to indicate the order of computation.

Suppose, for example, that data is being stored starting at MEM with a fifteen word introductory block, after which four words are used to store each entry of data. The position of the first words of the Nth entry could be defined as F(N) by the one line macro

```
DEFINE      F(N)
<           MEM+17+<4*<N-1>> >
```

However, arithmetical operations are performed in the conventional priority order, so one pair of angle brackets could be omitted. (Which pair?)

If subsequently we wanted to fill the second word of the Kth entry from accumulator AC, we could do it by

```
            MOVEM     AC,1+F(K)
```

(How would this instruction be *assembled*?)

Exercise: What if the number of the file entry were given by the contents of accumulator K?

A macro may be called with fewer arguments than its definition specifies. The assembler will provide a *default value*; this will be zero unless the programmer has specified other defaults in the DEFINE statement of the macro. You can see how defaults are specified in Figure 3.10; note that angle brackets must be used, and that intervening spaces are not permitted. So

```
            IREAD     MEM
```

would read one (default for number) integer base ten (default for base) into MEM. However, to read one number of base two into MEM, the call should be

```
            IREAD     MEM,1,2
```

because of the order in which the arguments must be given.

Labels within a macro could cause problems; the macro will be assembled in different places, and one label cannot serve to name several different locations. The assembler will take care of this problem for you if your labels within a macro have % as their first character. These labels must also appear in the DEFINE statement, although if they are placed after the other arguments they may be forgotten when the macro is called. The assembler will create special symbols, of the form .. followed by four digits, for each label as required (do not use symbols starting with .. yourself).

The macro can reference a label outside itself, as with ERR in the IREAD macro. Of course location ERR must appear somewhere, as it does in Figure 3.12 which, with Figure 3.11 which is discussed below, completes a program using the macro. (Note the several line literal at ERR. How do you suppose it is assembled?) It is best not to try from outside a macro to reference labels inside it.

Exercises: Write macros to meet the following specifications.

(i) JMPL X,Y,Z
jumps to the line labeled Z if the contents of location X are less than the contents of location Y.

(ii) IREAD2 X,N,B
similar to IREAD, except that now N and B are to be accumulators containing the required information.

(iii) FILCMP X,Y
compares word by word the contents of two blocks that start at X and at Y and end in each case with the first zero word encountered. If the blocks are identical in length and contents, the macro is to delete the entire block starting at Y.

```
;MACRO to print out N (default 1) integers, base B
;(default decimal), in C (default ten) columns, with
;S (default twelve) spaces between columns, from
;locations starting at X (default AC1).  Call by
; IPRINT  X, N, C, S, B    AC usage: 13 through 17

DEFINE  IPRINT (X<1>,N<1>,C<12>,S<14>,B<12>)
<       PUSHJ   P,PRINT
        X
        N
        C
        S,,0                        ;used at PRINT+2
        B       >

PRINT:  MOVE    17,P                ;rh AC 17 for referencing
        HRR     17,(P)              ;parameters.  lh for
        ADD     17,3(17)            ;spacing routine
        ADD     17,[1,,0]
        MOVEI   16,5                ;amend pushdown pointer
        ADDM    16,(P)              ;for return
        SETZB   13,16               ;column count,,number count
        CAML    16,1(17)
        POPJ    P,                  ;all done
        PUSHJ   P,PR1
        AOJA    16,.-3
PR1:    MOVE    14,(17)             ;routine to set
        ADD     14,16               ;next number
        MOVE    14,(14)             ;into AC 14
PR2:    IDIV    14,4(17)
        HRLM    15,(P)
        JUMPE   14,.+3
        PUSHJ   P,PR2
        SKIPA
        PUSHJ   P,PR3               ;to format routine
        HLRZ    15,(P)
        ADDI    15,60
        OUTCHR  15
        POPJ    P,

PR3:    SOJG    13,.+4
        OUTCHR  [15]
        OUTCHR  [12]
        MOVE    13,2(17)            ;spacing routine
        MOVE    14,P
        SUB     14,17
        OUTCHR  [40]
        AOBJN   14,.-1
        POPJ    P,
```

FIGURE 3.11 Macro to print out a number.

(iv) DO X,K
performs the subroutine starting at location X; decrements the contents of accumulator K by 1; and repeats if (K) > 0. Subject to:
(a) the subroutine may not change K;
(b) the subroutine may change K.
(Use a JSA to call the subroutine.)

The macro IPRINT in Figure 3.11 solves the problem of excessive space demands by large macros. The macro itself is merely the six line calling sequence for a subroutine. Note that the subroutine here is called by a PUSHJ, but the macro does not set up a pushdown pointer. This must therefore be done by any program that calls the macro.

There is otherwise nothing new in Figure 3.11. However, the address referencing requirements call for some rather delicate maneuvering. You should make a very careful line by line study of IPRINT .

A macro, like that in Figure 3.11, in which some lines consist merely of an argument to be passed, can cause problems if no default is specifically given for the argument, as under certain circumstances the assembler will generate no code at all for such lines. The safe way to pass a

```
        P=3
START:  MOVE    P,[IOWD 50,WRD]
        OUTSTR  [ASCIZ  /HOW MANY NUMBERS?  /    ]
        IREAD   INF+2
        OUTSTR  [ASCIZ  /HOW MANY COLS?  /       ]
        IREAD   INF+3
        OUTSTR  [ASCIZ  /HOW MANY SPACES PER COL?  /     ]
        IREAD   13
        MOVSM   13,INF+4
        OUTSTR  [ASCIZ  /BASE (UP TO ^D 10) FOR PRINT OUT?  /    ]
        IREAD   INF+5
        OUTSTR  [ASCIZ  /TYPE IN NUMBERS: /      ]
        IREAD   MEM,@INF+2
INF:    IPRINT  MEM
        OUTCHR  [15]
        OUTCHR  [12]
        OUTCHR  [12]
        JRST    START

ERR:    OUTSTR  [ASCIZ  /
INPUT ERROR!
/       ]
        JRST    START

MEM:    BLOCK   100
WRD:    BLOCK   50

        END     START
```

FIGURE 3.12 A program to illustrate the use of the macros in Figures 3.10 and 3.11.

parameter is instead of, for example

```
        JSA     16,ROUTIN
        X
```

specifying

```
        JSA     16,ROUTIN
        CAM     X
```

Note that CAM is a no-op, so some flexibility in the return instruction is achieved (in what way?). The line passing a parameter in this way assembles with index register and indirect bit clear, so no difficulties arise over calling a macro with arguments indirectly addressed or indexed. It may occasionally be relevant, however, that the left half of the word is not zero.

Other no-ops could be used, but in case of a return that "performs" the no-op passing the argument, the DEC official manuals warn for many of them that the address part is "reserved for future use, and should be zero." The ARG operator discussed in Section 4.2 may also be used.

Figure 3.12 binds the two macros together into a complete illustrative program. On assembly, the six locations starting at INF that comprise the macro IPRINT contain defaults for four of the parameters of this macro. But when the program is run, the IREAD instructions successively fill in these locations. Check this for yourself using DDT.

Exercises:

(i) Why is indirect addressing needed in the last IREAD call?

(ii) Eliminate the need to ask HOW MANY NUMBERS? by letting the program find out when you type them in. (Declare a special symbol to end input.)

(iii) Allow a choice of base for input also.

(iv) Rewrite IREAD as a short macro calling a subroutine.

(v) Note that a subroutine for which a macro is a calling sequence should not have the same name as the macro (above we used IPRINT for the macro and PRINT for the subroutine entry). Investigate why this is so. (Hint: consider the order in which MACRO searches symbol tables.)

Linking Files

If a subroutine or macro will be used in several different programs, we can write it as a separate file, and join it with each program at compilation time. For example, the macro in Figure 3.10 might form a file on its own; we could call it IREAD.MAC. Figure 3.11 might comprise a file called IPRINT.MAC, and the program in Figure 3.12 could be called TEST.MAC. The command EX TEST ↵ would fail, since IREAD and IPRINT would be undefined symbols. We join the macros to the program that calls them by giving the command

```
EX IREAD+IPRINT+TEST ↵
```

A similar command format may be used with COMPILE and LOAD. The files are linked together in the specified order at compilation, and the effect is exactly as if they had been written as one long file. So TEST must come last. (Why? Does the order of the other two matter?)

It does not take long before the experienced assembler language programmer collects a great miscellany of macros and subroutines performing a variety of tasks convenient to his or her needs. Eventually, for ease of reference, they should be collected together into a *library*. First we shall discuss how to do this with subroutines, and suppose that SBRTN1.MAC is typical of our subroutines. We suppose it is called by a JSA using accumulator 16, and that two arguments are passed.

```
SBRTN1:     0
            ...
            ...
            JRA         16,2(16)
```

Note that we may if we wish use the same name for the subroutine entry point as for the file that the subroutine comprises.

If SBRTN1 is to become part of a library, it must be distinguishable as a separate entity, and also be accessible from outside the library. A subroutine is made accessible to other programs by declaring its entry point to be such, in an assembler language ENTRY statement.

```
ENTRY       SBRTN1
```

This may appear anywhere in the file containing the subroutine; just before the first instruction of the subroutine is a convenient position.

To enable the subroutine to be loaded as a separate entity, an END statement should be provided. However, there should be no start address folowing END as an operand in the statement. The start address will be specified in the main program, not in a subroutine.

In addition, a TITLE statement will be convenient. In fact, it is convenient to have a TITLE statement in every program; this passes to the assembler the name to be given to the program. If there is no TITLE statement, the assembler will supply the name .MAIN , which is confusing if more than one file is being assembled. The name given in the TITLE statement need not be the name borne by the whole file in the directory. Later we shall see how one library file can contain many routines, all with different titles.

With these changes, the file SBRTN1.MAC now looks like this.

```
TITLE   SBRTN1
ENTRY   SBRTN1

SBRTN1: 0
        ...
        ...
        JRA     16,2(16)
        END
```

The TITLE statement may also appear anywhere in the file. Neither this nor the ENTRY statement generates any code on assembly.

Suppose now that we have prepared SBRTN1.MAC, SBRTN2.MAC and SBRTN3.MAC in this way, each in its own separate file with TITLE and ENTRY statements. Obtaining from these files

a single file, in a special library format, is a two stage process. Let us decide on the name LBRARY (six characters only!) for our library. We create a temporary file with this name by compiling our subroutines using the /FUDGE switch to the COMPILE command

```
COMPILE /FUDGE:LBRARY SBRTN1, SBRTN2, SBRTN3 ↵
```

Note the syntax of this command, in particular the symbol : before the file name given as argument to the /FUDGE switch. Note also that, although the subroutines are to be combined together into one file, they are separated in this command by commas, not by the + sign. These details are required with use of the /FUDGE switch, and must be observed.

So far all we have is a file containing information necessary to the creation of a library file. We actually create the file by giving as the next monitor command

```
FUDGE ↵
```

The FUDGE command creates a library file with the name given as argument to a /FUDGE switch in a COMPILE command. The FUDGE command must immediately follow the COMPILE command; any intervening command that runs PIP will destroy the necessary information.

We now have a disk file named LBRARY.REL containing our three subroutines in the specified order. To access one or more of these subroutines from a program, there must appear within the program a warning to the assembler that the subroutine entry points are not within the program itself. This is done by giving the names of the entry points as arguments in an assembler language EXTERN statement. For example, if, anywhere in a program, the following line appears

```
EXTERN SBRTN1, SBRTN3
```

then calls such as JSR SBRTN1 or JSA 16,SBRTN3 may be made in the program. However, a call to SBRTN2 will not succeed, even though this subroutine is in the same library file as the other two. Without an EXTERN statement the assembler will not look elsewhere for SBRTN3, which it will regard as simply an undefined symbol.

An alternative way of declaring a user defined symbol to be external to the program in which it is used is to suffix ## to the symbol the first time it is used. There must be no intervening spaces between the symbol and this suffix. Thus if there is no EXTERN statement for SBRTN1, it could nevertheless be called by

```
JSR     SBRTN1##
```

An EXTERN statement or ## suffix prepares the assembler to look elsewhere for a symbol. The assembler will not, however, hunt through your files for such a symbol on its own initiative: it must be told explicitly where it should look. This is done when the program is loaded or executed. Although SBRTN1 is still a separate file SBRTN1.MAC (incorporating an ENTRY statement), if TEST.MAC contains a call to this subroutine then the execution command should be

```
EX TEST, SBRTN1 ↵
```

Note that

```
EX SBRTN1+TEST ↵
```

would not work. SBRTN1 is no longer merely a separately written subroutine, ready to be fused into a program at compilation time. It is now a file in its own right, with its own END statement. The assembler would concatenate SBRTN1 and TEST, and stop assembly as soon as it encountered an END statement; this happens without TEST being assembled at all. Since the start address is in the END statement of TEST, the above execution command would receive the "no start address" error message from LINK-10.

If the END statement of SBRTN1 had been removed, concatenation would still not work. The assembler has been warned, by an EXTERN statement or ## suffix, that SBRTN1 is an *external* symbol. It will not, therefore, be satisfied with the similar symbol that it now finds *within* the program.

Once SBRTN1 has been incorporated into LBRARY, the same form of execution command could be used, but now LBRARY replaces SBRTN1. The result, however, is that the whole of the file LBRARY is loaded. This would be very wasteful if only a few routines in a large library file were being used. This is avoided by preceding the name of the library file with the /SEARCH switch in a load or execute command.

```
EX TEST, /SEARCH LBRARY ↵
```

This switch is also available with the DEBUG command. LINK-10 responds to the /SEARCH switch by loading from the library file only those routines specifically needed by the program. You should use DDT to check this effect for yourself.

Except for the specified ENTRY points, the same symbol may be used in the program and in various library subroutines, with a different meaning each time. The location MEM may be defined in two subroutines, but the assembler will provide two different locations, one for each subroutine. Unless specified otherwise, MEM is a *local* symbol, known only to the program or library routine in which it is defined. Thus it is no use referring to accumulator AC in a program, if AC=1 is defined only in a library subroutine; the assembler will regard the program's use of AC as a reference to an undefined symbol. This is normally a great convenience, lessening the need to invent endless new symbol names.

Sometimes, however, we may want to refer in a program to a symbol defined in a library subroutine, or vice versa. The symbol must then be declared available to other programs, or *global,* when it is defined. The ENTRY statement does this, but it also declares the symbol to be a library entry point, which could be misleading. To declare the symbol X global, we use the assembler language INTERN statement

```
INTERN X
```

Alternatively, if X is a location, instead of reserving it (with zero initial contents) by

```
X:      0
```

we can simultaneously reserve it and declare it global by

```
X::     0
```

If X is defined by equality with another symbol, as, for example

```
X=3
```

then it can be simultaneously defined and declared global by

```
X=:3
```

A program or subroutine referencing a symbol defined elsewhere as global must declare that symbol to be external, by an EXTERN statement or ## suffix.

The situation is somewhat different for macros because the code that is to substitute for the macro call must be within the program calling the macro. This can be achieved by concatenation at execution time. Our own approach is to have all our macro definitions in a single file called MACROS.MAC. Since these are all either very short routines or subroutine calling sequences of a few lines, the file is quite short. This file is concatenated at execution time (using the + construction) with any program calling any of our macros, so it has no END statement. The subroutines called by the macros are all in LBRARY.REL. We can execute any program, say, TEST.MAC, calling any of our macros, by the command

```
EX MACROS+TEST, /SEARCH LBRARY ↵
```

We find it convenient to use for the name of the entry point of a subroutine called by a macro the name of the macro itself preceded by the $ character. This is a dollar sign, not ESCAPE. Together with letters of the alphabet, $. and % may be the first or any other characters of a

symbol name; however, . and % have special uses as first characters in symbol names, and confusion could result from their routine use as such. Thus we could have in MACROS the definition

```
DEFINE      SWAP      (X,Y)
<           JSA       16,$SWAP
            ARG       X
            ARG       Y          >
```

although in practice we would not. (Why not?)

To see what your library contains, use the system program MAKLIB . System programs are run by the R command to the monitor

R MAKLIB ↵

MAKLIB issues the symbol * to indicate that it is ready to receive a command, and commands are entered with ↵. The commands to this system program have very fastidious syntax requirements, and should be entered exactly as shown below.

You can tell MAKLIB to list the individual routines, or modules, in your library file LBRARY.REL by commanding

TTY:=LBRARY/LIST ↵

The use of the mnemonic TTY indicates that the listing is to be typed out at the terminal. The entry points are obtained by

TTY:=LBRARY/POINTS ↵

The module SBRTN4.REL may be appended to the library file by

LBRARY=LBRARY,/APPEND SBRTN4 ↵

For this to work, SBRTN4.MAC must already have been written in the form specified above, ready for insertion into a library file, and *it must have been compiled.* MAKLIB operates only with files having the suffix .REL, but the suffix should not be given in the MAKLIB command.

To delete, say, SBRTN2 from LBRARY, issue to MAKLIB the command

LBRARY=LBRARY/DELETE:(SBRTN2) ↵

Exit from MAKLIB with ^C.

Exercise: Practice creating, updating, and using your own library of subroutines and macros. Be sure to keep a record of the function, calling sequence and AC usage of every module in your library.

CHAPTER FOUR

DATA MANAGEMENT

4.1 BYTES

Many programming tasks involve the storage, manipulation, and output of text, which will be handled within the computer in the form of ASCII code. Each ASCII code requires seven bits; but we cannot so far deal with anything smaller than half a word. Storing a long text at the rate of two characters per word would soon cause embarrassing difficulties over memory allocation. It is much more efficient to pack in ASCII characters five to a word, thereby using all but one bit of each word. This is the way in which text given in an ASCIZ statement is assembled. (Check this for yourself.)

A collection of adjacent bits within a single word is called a *byte.* Thus, ASCII codes occupy 7-bit bytes. (Byte is pronounced like bite.)

There is a special instruction to take a byte of any size from the rightmost possible position in an accumulator to any position in a memory word (*deposit* the byte); and one to take a byte from any position in memory to the rightmost possible position in an accumulator (*load* the byte). Observe that, for ASCII codes, INCHWL reads in a byte to the rightmost position in a location; so if that location is an accumulator, the byte is ready to be deposited. Correspondingly, a byte that has just been loaded is ready to be transmitted to the terminal by an OUTCHR instruction. Note that we do not consider collections of bits that straddle two words.

Before we can try to move bytes around, we must have a clear way of referencing the position of a byte. Somewhere we must have available the address of the location of which the byte is a part, the position of the byte within that location (perhaps in terms of the bit position of one end of the byte), and the size of the byte. This information must be stored in a particular specified format, and is then called a *byte pointer.*

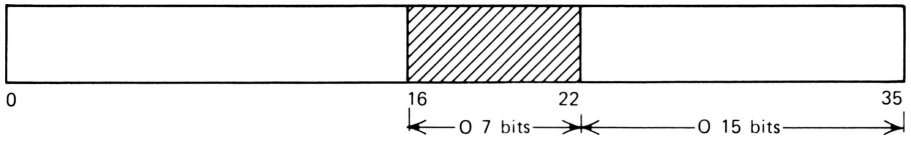
In the location chosen to house the byte pointer, bits 13 through 35 are reserved for the memory reference; this yields the address of the memory location in which the byte is to be found. The location housing the byte pointer (*not* the location housing the byte itself) is the memory reference in the byte manipulation instructions discussed below. The byte manipulation instructions perform the usual effective address calculation, and the final word retrieved in that calculation is taken to be the byte pointer. Once the byte pointer has been retrieved in this way, the location

housing the byte itself is determined by an effective address calculation on the contents of the byte pointer. This is performed just as if the byte pointer were an instruction.

Exercise: Suppose a byte manipulation instruction has indirect addressing in its memory reference. Can the byte pointer reference the location in which the byte is housed via indirect addressing?

The rest of the byte pointer defines the position of the byte within the word. Bits 0 through 5 of the byte pointer give the number of bits to the right of the byte in the word. Bits 6 through 11 give the size of the byte in bits. (Bit 12 should be 0.) For example,

points to the following byte in location WRD



Note that byte sizes have been given in *octal* notation, but bit positions are *decimal.*

Fortunately, a byte pointer can be set up without specific reference to the above word format, using the special assembler language POINT statement. POINT looks rather like an assembler language instruction, but in fact is merely a direction to the assembler to set up the contents of a word in a particular way. POINT is an example of a *pseudo-op;* so also are ASCIZ, BLOCK, DEFINE, IOWD, and END.

A byte pointer to the byte shaded in the illustration above would be set up in location MEM by declaring as an initial value

```
MEM:        POINT     7,WRD,22
```

In the POINT statement, the second argument is the address in which the byte is to be found; this may be indirect and indexed, with the consequences implied by our discussion above. The first argument is the size of the byte in bits, as a *decimal* number. The third argument is the bit position in the word of the rightmost bit of the byte, as a *decimal* number.

If the third argument is left unspecified, the default valve supplied by the assembler is the imaginary bit to the left of bit 0. This is then the rightmost bit of the previous location. At first sight this default seems perverse, but we shall see that it is quite convenient.

The above byte pointer could also be set up in a program by

```
MOVE        [POINT 7,WRD,22]
MOVEM       MEM
```

The following program is meant to be checked through using DDT since it produces no output. It reads ASCII text, packing it five characters to a word, starting from the leftmost 7-bit byte of location WRD. Notice that the byte pointer is set up to point to the byte *before* the first one to be used for storage.

```
        AC=1
START:  INCHWL  AC
        IDPB    AC,MEM
        JRST    START
MEM:    POINT   7,WRD
WRD:    BLOCK   100
        END     START
```

The instruction Increment the byte pointer and DePosit the Byte

```
IDPB        AC,MEM
```

(i) increments the contents of MEM to produce a byte pointer that points to the next adjacent byte of the size specified in the pointer; if there is insufficient room in the current word pointed to, the leftmost such byte in the next word is referenced;
(ii) deposits the rightmost byte of the specified size from AC into the position *now* referenced by the byte pointer.

The contents of AC, and of the remaining bits in the destination word, are unchanged.

Thus, the first IDPB instruction in our program modifies the pointer to reference the first byte reserved for storage; only then does it deposit into it the byte from accumulator AC.

Exercises:

(i) Run this program with DDT. Check the contents of MEM and of the block starting at WRD after every IDPB instruction.

(ii) The instruction DPB DePosits a Byte, but does not affect the byte pointer. Use it to amend the above program so that the symbol # is not entered as text, but rather "erases" the last character typed by replacing it with the next character to be typed.

*(iii) Develop # into a proper "erase key." Each # should cause the byte that the pointer is referencing to be replaced by the null byte (all zeros), and move the pointer back to point to the previous character. The null characters will be "typed over" if there is further input. Allow a special character to signify end of input, and insert a null byte at the end of the text.

Note that the ASCIZ statement assembles text with at least one null byte at the end. If the last word used for the text is not wholly filled by it, the remaining bytes are set to zero. This alone would not guarantee a null byte at the end of the text, as the number of characters might be a multiple of five; in this case, a whole null word is appended to the text. Thus, ASCIZ suits the action of OUTSTR , which stops output when it encounters a null byte.

The instruction IBP will increment the byte pointer by one byte of the size specified in the pointer, without moving any data. Thus, there is no use for the accumulator field, and it should be zero. Only on the KL10 processor is there an instruction, which we discuss below, able to decrement the pointer as required by Exercise (iii) above. So you are called on to use your ingenuity in that exercise!

Alphabetical Ordering

To illustrate the problems arising in the manipulation of data stored in bytes, we shall develop a program to read a list of text words (by this we mean words in the usual linguistic sense, rather than computer words), and print them out in alphabetical order.

As an approach to this task, observe that alphabetical order of individual letters corresponds to increasing numerical order of their ASCII codes. So we can develop the idea of comparing adjacent text words along the same lines as we compared numbers in adjacent computer words in the program of Figure 2.6 in Section 2.3. You should review that program before reading on.

Suppose that in locations BP1 and BP2 we have byte pointers referencing the bytes preceding the first letters of two adjacent text words. Suppose also that a space indicates that a text word has terminated (distinguish this from the null character, which ends the whole text). Let us write a routine to check whether the text word referenced via BP1 should come before the other one in an alphabetical ordering. We must check the text words, letter by letter; let us carry out the check by loading bytes into accumulators CH1 and CH2. We perform the following steps:

```
CHECK:  ILDB    CH1,BP1         ;step (i)
        CAIG    CH1,40          ;step (ii)
        JRST    NOSWAP
        ILDB    CH2,BP2         ;step (i)
        CAIG    CH2,40
        JRST    SWAP            ;step (iii)
        CAMLE   CH1,CH2
        JRST    SWAP            ;step (iv)
        CAMN    CH1,CH2
        JRST    CHECK           step (vi)
NOSWAP: ...                     ;step (v)
```

FIGURE 4.1 Routine to check the alphabetical ordering of two text words.

(i) increment the byte pointers and load fresh bytes from each text word;
(ii) if CH1 contains a space, we do not swap the text words (this will ensure that, for example, THIN will precede THING);
(iii) if CH2 contains a space or is null, we must swap;
(iv) if (CH1) > (CH2), we must swap;
(v) if (CH1) < (CH2), we do not swap;
(vi) if (CH1) = (CH2), go to step (i).

Since space has ASCII code O 40, less than that of any letter of the alphabet, step (ii) could be incorporated into step (iii). (How?) However, it will be more convenient for later development to keep these steps separate.

A routine to check the alphabetical ordering of two text words appears in Figure 4.1. We have used the ILDB instruction to Increment the byte pointer and LoaD a Byte. The action of this instruction parallels that of IDPB. The pointer is first incremented, in exactly the same way, then the byte is moved. Remember that the byte always goes to the rightmost position in the referenced accumulator; in this case it occupies the seven lowest order bits. The rest of the accumulator is set to zero (contrast this with the action of IDPB on its destination).

Routine NOSWAP in Figure 4.1 might be simply a return to the mainstream of the program.

The way in which we have stored text would make routine SWAP rather awkward. Suppose our text was THE QUICK BROWN FOX. The first two text words store as

T	H	E	#	Q
U	I	C	K	#

where # represents a space. These text words must be swapped, and stored as

Q	U	I	C	K
#	T	H	E	#

Exercise: Write a suitable SWAP routine. You may find the instruction LDB helpful; it will LoaD a Byte, without affecting the byte pointer.

The above exercise should have persuaded you that shifting bytes around in memory is tricky and liable to error. We can avoid it in this instance by an approach that has wide applicability. We store our text in a block of memory (starting, say, at WRD) with no indication as to its structure. No character is kept in this block to indicate the end of a text word, or even of the whole text. We illustrate this on the left of Figure 4.2.

```
WRD:  | T | H | E | Q | U |        LIST:  |  3  |  1  |
      | I | C | K | B | R |               |  5  |  4  |
      | O | W | N | F | O |               |  5  |  9  |
      | X |   |   |   |   |               |  3  | 14  |
                                          |  0  |  0  |
```

FIGURE 4.2 Storing the structure of a text separately from the text itself.

Now, in a block beginning at, say, LIST, we keep a record of the structure of our text. In the right half of each computer word we record the byte number, counting from the leftmost byte in WRD, at which our given text word starts; in the left half will be the number of letters in the text word. This is illustrated on the right of Figure 4.2 (decimal notation). For example, at byte number nine a text word of five letters starts: it is BROWN. A null word at the end of this block indicates that there is no more text to be referenced.

Initially, the block at LIST will be set up to hold the structure of the text as it is read in. But now we can change the structure by manipulating the block at LIST, without doing anything at all to the text storage block at WRD. For example, swapping the first two text words is accomplished by interchanging the contents of LIST and LIST+1.

Exercises:

(i) With this method of storage, is there an easy way to delete or insert text words?

(ii) Amend routine CHECK for this method of storage. You may assume that BP1 and BP2 have already been set up to point to the bytes preceding those given by the right halves of LIST(N) and LIST+1(N). Move the letter counts from the left halves into accumulators CT1 and CT2.

*(iii) Write a routine to set up a byte pointer to reference the byte before that given by the right half of LIST(N).

*(iv) Write a program to accept a list of text words typed in at the terminal, and print them out in alphabetical order. Setting up the block at LIST requires care.

(v) Write a program that will read text typed in at the terminal, regarding all letters as lowercase, whether entered as upper or as lowercase. (Your terminal must have lowercase capabilities for this.)

(vi) Amend the above program so that the symbol ^ causes the next character, if it is a letter, to be entered as upper case; the symbol ^ itself should not be entered in the text.

(vii) Write a program to read ASCII text, allowing *eight* bits per character. The extra bit is to be set to 1 if the symbol & precedes the character (the symbol & itself should not be entered in the text). Now have your program type out the text, underlining characters in which the extra bit is set to 1. Underlining requires use of the ^M character, and some careful counting. (Your terminal must have the _ character for this; on some it is replaced by the ← character.)

(viii) Amend your program of Exercise (iv) above to accept both capitalized and noncapitalized text words.

On the KL10 processor an instruction is available to ADJust the Byte Pointer. The instruction

```
ADJBP    AC,MEM
```

retrieves a byte pointer from location MEM in the usual way. It then adjusts this pointer by the number of bytes, positive or negative, given by the contents of AC. If the contents of AC are positive, the pointer is moved forward the appropriate number of bytes; negative contents of AC move it backward. The new byte pointer is placed in AC. The contents of MEM are unaffected.

In this instruction AC must not be accumulator 0. The instructions ADJBP and IBP both have operation code O 133, but are distinguished by the AC field. The use of distinct mnemonics helps to avoid confusion between these rather disparate instructions.

Logical Instructions

A very powerful way of manipulating bits within a word is given by the *logical instructions*. The reason for this terminology is that, if we interpret a 1 in a bit as representing "true" and a 0 as representing "false," then these instructions perform logical operations corresponding to their mnemonics. For example, if the letters p and q represent statements, then in logic the combined

statement "*p* AND *q*" is true precisely when both *p* and *q* are true. So the instruction

```
AND     AC,MEM
```

forms the AND function of the contents of AC and MEM; this has a 1 in a bit precisely where both AC and MEM have a 1. The destination for the result is AC in the basic mode, as above; modes ANDM and ANDB use Memory and Both as destination. For example, suppose that

```
(AC) = O 26 = B 10 110
(MEM) = O 13 = B  1 011
```

The only bit that is set to 1 in both AC and MEM is bit 34. So after

```
ANDM    AC,MEM
```

AC still contains O 26, but MEM contains O 2.

The Immediate mode allows an AND function with a word whose right half is the given number, and whose left half is zero. For example,

```
ANDI    AC,17
```

sets to zero all but the lowest order four bits of AC; these are unaffected.

Since each bit can represent a value "true" or "false" the AND instruction actually performs thirty-six logical operations simultaneously. The reader who is unfamiliar with formal logic can regard the logical operations merely as instructions producing certain specified bit patterns. However, at some stage it will be helpful to acquire a basic knowledge of the elements of symbolic logic. This is especially true for programmers using those higher level languages, like FORTRAN and ALGOL, in which logical operators (such as IF) are specifically available.

To specify the result of a logical operation, we need to know what it does for all possible bit configurations in AC and MEM. For each bit we must consider the four possibilities: both AC and MEM set to 0; both set to 1; AC set to 0, MEM set to 1; AC set to 1, MEM set to 0. We can therefore define such an operation by a *configuration table*. For AND the configuration table is

AC	0	0	1	1
MEM	0	1	0	1
AND	0	0	0	1

Together with AND, the basic logical operators are OR and NOT. The OR operator requires care, because in everyday speech it is not always clear whether "or" means *inclusive* or:

"*p* IOR *q*" is true exactly when *p* or *q* *or both* is true

or whether it means *exclusive* or:

"*p* XOR *q*" is true exactly when *p* or *q* *but not both* is true

The configuration tables should be compared.

AC	0	0	1	1
MEM	0	1	0	1
IOR	0	1	1	1
XOR	0	1	1	0

The mnemonic OR may be used instead of IOR.

All sixteen logical instructions are listed in a configuration table in Figure 4.3. Each instruction is given in its basic mode. For each of them, the other modes may be obtained by appending I, M, or B.

The logical operator NOT is represented by forming the *logical complement* of the contents of the word: all 1's are set to 0, and vice versa. So ANDCM is mnemonic for AND with the Complement of Memory. The equivalent logical statement is "*p* AND (NOT *q*)." Similarly ORCB is mnemonic for OR with the Complement of Both, equivalent to "(NOT *p*) OR (NOT *q*)." Note that OR is always inclusive unless specified otherwise.

	AC	0	0	1	1
	MEM	0	1	0	1
SETZ		0	0	0	0
AND		0	0	0	1
ANDCM		0	0	1	0
SETA		0	0	1	1
ANDCA		0	1	0	0
SETM		0	1	0	1
XOR		0	1	1	0
IOR		0	1	1	1
ANDCB		1	0	0	0
EQV		1	0	0	1
SETCM		1	0	1	0
ORCM		1	0	1	1
SETCA		1	1	0	0
ORCA		1	1	0	1
ORCB		1	1	1	0
SETO		1	1	1	1

FIGURE 4.3 Configuration table for the logical instructions.

Exercise: (i) Why is there no XORCM instruction?

(ii) In the sequence

```
SETCMI    AC,X
JUMPGE    AC,LABEL
```

when does the second instruction cause a jump to LABEL?

(iii) Which instruction interchanges the ASCII codes for the + and − characters?

(iv) Given any two characters, is it always possible to find a single logical instruction that interchanges their ASCII codes?

*(v) Write a program that will successively request input of: the mnemonic for a logical instruction, in basic mode; the contents of AC; the contents of MEM. The program should then type out the contents of AC resulting from implementation of the given logical instruction. (Hint: make a table of the mnemonic codes

```
LIST:     ASCII     /SETZ/
          ASCII     /AND/
```

and so forth. The ASCII statement differs from ASCIZ in that the former will not supply a null word to guarantee a null byte at the end of text. It is needed here because some of the mnemonics are five letter codes. Now, when your program reads a mnemonic, check through the list at LIST until it is found. Then perform the appropriate one of a list of routines.)

Parity

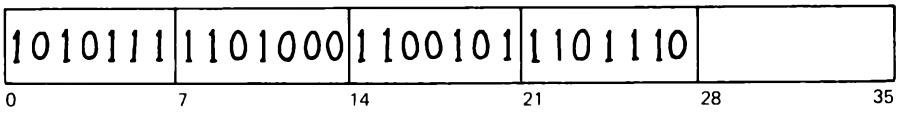
We shall use the XOR and AND instructions to illustrate one way of dealing with a recurrent problem arising whenever quantities of information are stored in a computer or on magnetic tape: error. A damaged disk or tape might have bits here and there set to 1 when they should be 0, or vice versa, and there might be no way of knowing this. Even if it eventually became clear that something was amiss, it might be very difficult to track the problem down to one particular source. It is plainly a good idea to build in a check on the veracity of stored information.

One way of doing this uses *parity*. Suppose that information is to be stored in 7-bit bytes, as, for example, with ASCII text. For each data item we reserve, in addition to the seven bits needed to house it, an extra bit that we shall call the *parity bit*. We must decide in advance whether we want every 8-bit byte to have odd or even parity: that is, to contain an odd or an even number of 1's. The decision is wholly arbitrary; we shall opt for even parity here.

An example will help. Suppose that we are storing the information: *When*. The ASCII codes are

W	O 127 = B 1 010 111
h	O 150 = B 1 101 000
e	O 145 = B 1 100 101
n	O 156 = B 1 101 110

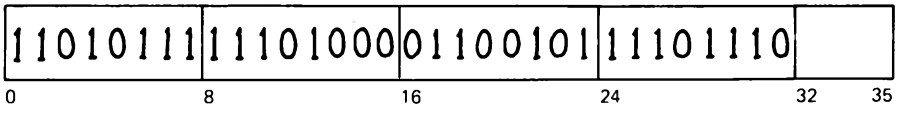
Count the number of 1's in each code. The byte representing *W* has five 1's: this is odd parity. Likewise the bytes representing *h* and *n* have odd parity; but that representing *e* has even parity since its ASCII code contains four 1's. As 7-bit bytes, *When* would store as



To incorporate a parity check, we use eight bits for each ASCII code. In the leftmost of these bits we put a 1 or a 0 to make the total number of 1's—and hence the parity—even. So the individual letters will be stored as

W	B 1 1 010 111
h	B 1 1 101 000
e	B 0 1 100 101
n	B 1 1 101 110

and the whole word *When* will store as



Now consider what happens when the stored information is read. We load 8-bit bytes into an accumulator, and check each time that the parity of the byte is even. If it is, we ignore the leftmost (parity) bit, and read the rightmost seven bits. Observe that OUTCHR transmits only the rightmost seven bits in a word. On the other hand, if the parity is odd, the program can warn us that something is amiss.

A parity check will fail in its purpose only if every damaged byte has been left with its parity nevertheless unchanged; that is, if an even number of bits have been affected. As it is not very likely that this will be the case in every byte, the check is a good one. In the DECsystem-10 itself, the memory locations are 37-bit words. Thirty-six of these bits are available to the user; the remaining bit is set so that the whole word has odd parity. This provide an internal check for the system itself, rather than for individual programs.

With ASCII text, there is a natural breakup of information into bytes. There is, however, no reason why any kind of data to be stored should not be broken into bytes, with an extra parity bit provided for each.

For 7-bit bytes, inclusion of a parity bit increases storage space consumption by 25% because only four bytes, rather than five, can now fit into a word. It also takes time to set the correct parity on input, and to check it on output.

We shall devise a way of working out the parity of an 8-bit byte. This routine can then be used in a program to set parity, as well as in one to check it. The contents of an 8-bit byte, regarded as a

binary integer, can vary in value from

B 00 000 000 = D 0

to

B 11 111 111 = O 377 = D 255.

One approach is to create a D 256 line table, with each line containing the instruction to be performed if the byte under consideration has the corresponding value. First we would need to know the parity of the binary representation of each integer between 0 and O 377. The first O 20 of these are

O 0 = B 0 even	O 10 = B 1000 odd
O 1 = B 1 odd	O 11 = B 1001 even
O 2 = B 10 odd	O 12 = B 1010 even
O 3 = B 11 even	O 13 = B 1011 odd
O 4 = B 100 odd	O 14 = B 1100 even
O 5 = B 101 even	O 15 = B 1101 odd
O 6 = B 110 even	O 16 = B 1110 odd
O 7 = B 111 odd	O 17 = B 1111 even

Now we create an instruction table. We suppose that subroutines EVEN and ODD have been written to deal with the respective parities. The first few lines of our table will then be

```
TABLE:  JSR     EVEN
        JSR     ODD
        JSR     ODD
        JSR     EVEN
        JSR     ODD
```

and so on.

Suppose the byte currently under consideration has been put in accumulator AC. Most likely this has been effected by an LDB or ILDB instruction, so that the byte is right justified in AC, and the remaining bits of AC are set to zero. Then it is the contents of AC that tell us which instruction in the table starting at line TABLE should be carried out. For example, if the byte were 00 000 010, then the contents of AC, regarded as an integer, have value 2. In this case, the instruction to be carried out is the JSR ODD found at line TABLE+2. We may generalize this to the following statement: *the instruction to be carried out is at line* TABLE(AC).

The question remains: having moved our byte into AC, how do we now ensure that the next instruction to be carried out is the one at line TABLE(AC), without at the same time losing the flow of the program? For example, if the next instruction in the program were JRST TABLE(AC) then indeed the instruction found at TABLE(AC) would be carried out, as desired. But the JSR at that location would have control returned to the line following itself in the table, not to the next line of the program; this would produce nonsensical results.

For our purposes here, we need an instruction whose function is to carry out the instruction to be found at a given location, but without altering the sequential flow of the program. Such is the eXeCuTe instruction

```
XCT     MEM
```

where MEM may be indexed or indirectly addressed. The AC field in this instruction should be zero. An XCT will perform any instruction found at the location given by its effective address calculation. It will even perform another XCT, and so on without end! The crucial point for our program, however, is that an XCT does not transfer operations to the location of the instruction to which it refers. It may be helpful to think of an XCT as rather replacing itself by the instruction found at the referenced location. If, in our program, we issued the instruction

```
XCT     TABLE(AC)
```

with AC containing 2, then the effect would be to perform the instruction JSR ODD at line TABLE+2. When the subroutine returns with the normal JRST @ODD however, the line it returns to is the one that follows the XCT.

Exercise: At which line is the instruction that will be performed following the sequence

```
        XCT     LAB
LAB:    SKIPA
        ...
```

Can you suggest a simpler way to produce the result of this instruction sequence?

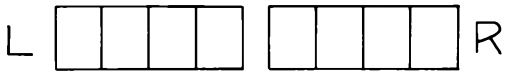
We now have an easy way of dealing with parity. The single instruction XCT TABLE(AC) references the correct line of the table. In some cases the necessary action is provided by just one instruction; this can be put in the table, and there is then no need to jump to a subroutine. For parity setting, if the parity is odd, then the leftmost bit in the 8-bit byte needs to have its value changed. This can be done by eXclusive OR with B 10 000 000, so in the table every JSR ODD would be replaced by

```
        XORI    AC,200
```

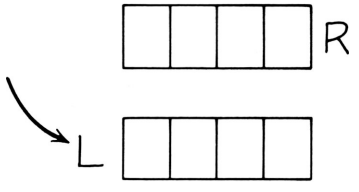
If the parity is already even, then nothing needs to be done. The XCT still, however, references the table, and so in place of any JSR EVEN it must find there an instruction that specifically does nothing at all—in other words, a no-op.

For checking parity, JSR EVEN in the table can again be replaced by a no-op. The JSR ODD instructions would remain, with ODD being the starting line of a subroutine for dealing with erroneous data.

Although this is quite a simple procedure, the table required takes up an excessively large amount of space. This can be overcome, at the cost of some extra computation. We are dealing with 8-bit bytes. Suppose we divide each byte into two 4-bit bytes



shift one alongside the other



and now eXclusive OR these 4-bit bytes together. The resulting 4-bit byte will have a 1 wherever either the left half or the right half (but not both) of our original 8-bit byte had a 1; but if two 1's have come alongside each other, XOR will lose them both and display a 0. In other words, XOR disposes of 1's in our two 4-bit bytes *only in pairs*. It follows that the resulting 4-bit byte has the same parity as the original 8-bit byte. This yields a great saving in table space. The value of the contents of a 4-bit byte, regarded as an integer, lies between zero and fifteen. Thus, the above first sixteen lines of the table are all that is required.

It remains to generate the required 4-bit byte from an 8-bit byte in AC. First, copy AC into, say, accumulator N

```
        MOVE    N,AC
```

then shift N so that the left four bits of the byte in N come alongside the right four bits of the byte

in AC

```
LSH     N,-4
```

and form right justified in AC a 4-bit byte of the same parity as the original 8-bit byte

```
XOR     AC,N
```

Now discard the rest of AC

```
ANDI    AC,17
```

and reference the table

```
XCT     TABLE(AC)
```

Exercises: (i) Write a complete program to store in a block of memory text typed in at the terminal, incorporating a parity bit for each 7-bit ASCII character, using
(a) even parity;
(b) odd parity.
(ii) Write a complete program to type out text stored as in the previous exercise, while checking the parity. Have your program respond to incorrect parity by printing some special symbol next to the suspect character, or by underlining it.
(iii) Join the above two programs together, with an intermediate stage that "damages" the text. Use your ingenuity to achieve this!

4.2 ARITHMETIC

In Section 2.1 we learned how to perform the basic arithmetical operations on integers, as long as the result would always fit into a computer word. In Section 3.3 we briefly considered how to deal with overflow.

This section introduces the facilities of MACRO-10 for handling larger numbers, and not necessarily integral quantities. We can do no more here than scratch the surface of a vast subject, of which full understanding depends on a substantial level of mathematical competence. Our purpose in this section goes little further than making the arithmetical instructions of MACRO-10 available to the student who has learned or will learn the principles of computational arithmetic elsewhere. We also presuppose greater familiarity with binary and octal arithmetic than is needed in the rest of this book.

As we observed in Section 3.3, the ADD and SUB operations are accurate for all operands, to the extent that the correct result can always be deduced from the result obtained, together with the state of the flags. IDIV also loses no information, since the remainder is preserved (as long as the destination is AC). To avoid losing information when multiplying, however, we must use not the Integer MULtiply instruction, but rather the MULtiply instruction. This is available in modes: basic, Immediate, to Memory, and to Both. First, this instruction calculates the product of the contents of AC and MEM in internal registers. This product may contain up to seventy bits, plus a sign bit. This is then separated into a *high order word* and a *low order word.* The low order word is set to contain the thirty-five least significant bits, and its sign bit is set by the sign of the entire product. The high order word contains the remaining bits, right justified; its sign bit also conveys the sign of the entire product.

The instruction

```
MUL     AC,MEM
```

places the high order word in accumulator AC, and the low order word in accumulator AC+1. (If AC is accumulator 17, the low order word goes into accumulator 0.)

The instruction

```
MULM    AC,MEM
```

places the high order word in location MEM, and *loses the low order word* (even if MEM is actually an accumulator).

Regarding treatment of AC and MEM, MULI behaves like MUL. MULB treats AC as MUL does, and MEM as MULM does. (See also Exercise (ii) below.)

Exercises: (i) What is the effect of MUL 1,3 when accumulators 1 and 3 contain:
(a) 1,,1 and 400 000;
(b) 1,,1 and 2,,2;
(c) 1,,−1 and −1,,1;
(d) 530 000 and 600 000;
(e) 1,,−1 and 1,,1;
(f) both 1,,1.
(ii) What is the effect of MULB 1,2 when accumulators 1 and 2 both contain 1,,1?

MUL will set OVERFLOW only if both operands are equal to -2^{35}; the result stored is -2^{70}, with the correct magnitude but the wrong sign. (On the KA10 processor, an accumulator operand of -2^{35} is always treated as if it were $+2^{35}$ in this instruction, producing the wrong sign in the product.)

MUL is often used in calculations where the final result will be contained within a single word, but intermediate stages might overflow. It is complemented by the divide instruction DIV, which divides a double length number by a single length number, to produce a single length quotient. For this instruction to work properly, the dividend must be in the high order word/low order word form produced by a MUL instruction.

The instruction

```
DIV     AC,MEM
```

divides the double length number in AC and AC+1 by the contents of MEM. The single length quotient is placed in AC, and the remainder in AC+1.

Modes I, M, and B are available. When memory is specified as the destination, the remainder is lost, just as with IDIV.

The above description depends on the assumption that the quotient can be housed in one word. This will not be the case if the high order word of the magnitude of the number in AC and AC+1 is greater than or equal to the magnitude of the contents of MEM. Under such circumstances no operation is performed, and OVERFLOW and NO DIVIDE are set in FLAGS. This could be dealt with by shifting one of the operands in the appropriate direction until DIV can work (remembering to shift the result afterwards; so the amount of shift must be recorded).

The single length divisor could be shifted by ASH. The double length dividend requires Arithmetic SHift Combined, ASHC. This is a shift on the 70-bit combination of all but the sign bits in AC and AC+1. The sign bits are not shifted; but the sign bit in AC+1 is set equal to the sign bit in AC (if AC,AC+1 have been set up by MUL, the sign bits are already equal). Note that with this approach some significant figures will be lost, unless we deal very carefully with the remainder.

Exercises: (i) Write routines to:
(a) approximate the double length result of dividing a double length number by a single length number (do it exactly if you can);
(b) negate a double length number;
*(c) print out a double length number in decimal notation. (Hint: 2^{35} = D 34 359 738 368 .)
(ii) Can ASHC set flags?
(iii) The logical instruction counterpart of ASHC is LSHC. Determine its effect. Can it set flags?
(iv) Determine the effects of the ROTate instruction ROT AC,X and the ROTate Combined instruction ROTC AC,X.

Until now we have discussed only *fixed point* arithmetic. The "point" referred to is the *radix* point; radix here has the same meaning as base. In radix, or base, ten, the radix point is the familiar decimal point. Just as columns to the left of the decimal point indicate successive multiplications by ten, so columns to the right of the decimal point indicate successive divisions by ten. Thus, D 12.34 denotes 1 × (ten) + 2 × (one) + 3 × (one tenth) + 4 × (one tenth of one tenth).

The same notation will serve in any base, with the value of the base replacing ten everywhere in the above description. Thus, O 12.34 denotes 1 × (eight) + 2 × (one) + 3 × (one eighth) + 4 × (one eighth of one eighth) = 8 + 2 + 3/8 + 4/64 in decimal notation. Here we are using an *octal* point.

Likewise, we may use a *binary* point. For example

$$\text{B } 0.\dot{1}\dot{0} = \text{B } 0.101010\ldots = \text{D } 1/2 + 1/8 + 1/32 + \ldots = \text{D } 2/3$$

(sum the geometric series).

The radix point here is *fixed* in the sense that, if there is one, the programmer knows exactly where it is assumed to be. Multiplication of D 123456 by D 234567 could be handled by the same machine instructions as multiplication of D 12345600 by D 0.0234567. The machine is in fact multiplying the integers comprising the significant figures of the operands. Elsewhere the programmer would be keeping a record of the position of the radix point. In the above example we have a decimal point. If ASH or ASHC is used to reposition numbers within computer words, the position of a binary point is affected.

Figure 4.4 contains a routine for reading a number in decimal notation, while keeping track of the decimal point. The significant figures will be housed, as an integer, in accumulator 1. Accumulator 3 will hold the negative of the number of integers following the decimal point. Thus, D 123.45 will appear as D 12 345 in accumulator 1 and −2 in accumulator 3. We are essentially imitating *floating point representation,* in which each number carries with it a record of the position of

```
        SETZB   1,3
        INCHWL
        CAIG    40          ;ignore leading separators
        JRST    .-2
        CAIN    60          ;ignore leading zeros
        JRST    .-4
        CAIE    56          ; . before significant figures
        JRST    INT
        INCHWI
        CAIN    60
        SOJA    3,.-2

FRACT:  CAIGE   60
        JRST    DONE
        SUBI    60
        IMULI   1,12
        ADDM    1
        INCHWL
        SOJA    3,FRACT

INT:    SUBI    60
        IMULI   1,12
        ADDM    1
        INCHWI
        CAIL    60
        JRST    INT
        CAIE    56
        JRST    DONE
        INCHWI              ;first char after .
        JRST    FRACT

DONE:   IDIVI   1,12
        SETZM   2
        ADJA    3,.-2
        IMULI   1,12
        ADD     1,2
```

FIGURE 4.4 Routine for reading a number containing a decimal point.

its radix point. This is very familiar from the use of exponents, particularly for expressing numbers of very large or very small magnitude in scientific calculations. Note that D 123.45 = D 12 345 × 10^{-2}. Check that accumulator 3 always holds the exponent.

Exercises:
(i) Study Figure 4.4, and draw a flow chart for it.
(ii) What is the purpose of the five lines starting at DONE ?
(iii) The routine does not satisfactorily handle input of zero (or of numbers that overflow, leaving zero behind: such as?). Correct this.
*(iv) Write a routine to read two numbers in this format, and to form, in the same format:
(a) their sum;
(b) their product, with as many significant figures, properly rounded off, as will fit into a single word.

Floating Point Instructions

MACRO-10 also has instructions for handling numbers in a special *floating point format*, in which the significant figures and the exponent are housed in separate fields within a single word. In the KI10 and KL10 processors, the floating point instructions are part of the *hardware*; that is, they are performed directly in special registers within the circuitry of the central processor. The KA10 processor may have no floating point facilities at all, or it may have the optional floating point package. In the latter case, some instructions are performed partly by the hardware and partly by the *software*; that is, by interpretive routines external to the basic circuitry. (The monitor, the various translators, and the assembler are all examples of software.) Other instructions are simply not available on the KA10, and will result in an error message if used in a program run on this processor.

It is so easy to perform complex arithmetical operations with the floating point instructions that many programmers pay little attention to the internal mechanism of the calculations performed by these instructions. This attitude is highly inadvisable, since floating point arithmetic is inherently approximate; only by acquiring a good understanding of the process can the programmer know the extent to which the results obtained are at all trustworthy.

The floating point representation of a number within a word is in three parts. Bit 0 gives the sign in the usual way. Bits 9 through 35 represent the significant figures as a binary fraction; you can imagine a binary point preceding bit 9. In decimal notation, it is as if we represented D 123.45 as D .12345 × 10^3. The same number could, however, also be represented as D .012345 × 10^4, and so on indefinitely. The first form, characterized by having the first numeral after the radix point not zero, is called the *normalized* floating point representation of the number. The floating point instructions are designed to work best with numbers in normalized binary floating point representation, and yield their results in that format also.

As an example, let us determine the fractional part of the normalized binary floating point representation of the integer D 3. In decimal notation 3 = 3/4 × 2^2; the exponent part of the representation is determined by the 2^2. The fractional part is

$$\begin{aligned} \text{D } 3/4 = \text{D } 1/2 + 1/4 &= \text{B } 0.1 + 0.01 \\ &= \text{B } 0.11 \end{aligned}$$

Exercise: What is the fractional part of the normalized binary floating point representation of the decimal numbers:
(a) 2; (b) 2.5; (c) 1/3; (d) 10; *(e) 1/10.

Decimal floating point numbers may be passed to the assembler in the form of a string of decimal digits including a decimal point. If the number to be represented in floating format is actually an integer, it should be followed by a decimal point and at least one zero. A scale factor may be provided by appending the letter E followed by a decimal integer (which may be negative or

zero). For example,

```
MOVE    AC,[-2.13E-3]
```

sets in accumulator AC the normalized binary floating point representation of D −0.00213. DDT will also accept type-in of floating point decimal numbers in this way, and you can scrutinize the effect by varying the type-out mode. Use this to check your answers to the above exercise. In addition, use various inputs to check the number of significant figures that the fractional part can accurately hold in floating point representation.

Bits 1 through 8 represent the binary exponent. For positive floating point numbers, the actual quantity housed in bits 1 through 8 is O 200 more than the exponent. For example, O 2.0 = $2^2 \times$ B 0.1 in normalized binary floating point representation. Since the number is positive, bit 0 is set to 0. In the fractional part, bit 9 is set to 1, and all the rest are 0. The exponent is 2, so the exponent field will contain O 202; thus bits 1 through 8 are set to B 10 000 010.

Again, consider D 1/8 = O 0.1 = B 0.001 = $2^{-2} \times$ B 0.1 in normalized form. The number is positive, so bit 0 is set to 0. The fractional part is as before, with a 1 in bit 9 and zeros elsewhere. And now the exponent is −2, so the exponent field will contain O 176; bits 1 through 8 are set to B 01 111 110.

An exponent of D −128 is represented by all zeros in bits 1 through 8; this is the negative exponent of largest possible magnitude. The largest possible positive exponent is D 127, represented by all 1's in bits 1 through 8.

The floating point representation of a negative number is the twos complement of the representation of its magnitude. Thus, it has a 1 in bit 0; in bits 1 through 8 it has the logical complement of the expression: O 200 + the exponent; and in the remaining bits it has the twos complement of the fractional part. For example, D −1/8 is represented by 1 in bit 0, B 10 000 001 in bits 1 through 8, 1 in bit 9, and zeros elsewhere.

The normalized floating point representation of zero is a word of all zeros.

Exercise: What is the normalized binary floating point representation of the decimal numbers: (a) 20; (b) −1/3; (c) −8; (d) 1025; (e) −1/5.

Clearly, arithmetical operations on floating point numbers must be carried out by instructions that take the floating point format specifically into account. Consider, for example, how the sum D 27 + 3 is formed, when these quantities are stored as floating point numbers. Their normalized binary floating point representations are, using octal notation for the fractional parts: D 27 = O 33 = $2^5 \times$ O 0.66; D 3 = $2^2 \times$ O 3/4 = $2^2 \times$ O 0.6. First, the representation of the number with smaller exponent is amended to make the exponents match. In this case, the result is to represent D 3 as $2^5 \times$ O 0.06. Of course this is no longer a normalized representation. Now the addition is performed:

$$2^5 \times \text{O } 0.66 + 2^5 \times \text{O } 0.06 = 2^5 \times \text{O } 0.74$$

The addition is in fact performed in double length. We shall not examine the details of the format of the double length result (which is in any case not made available to the user in its entirety), but merely point out that at this stage no significant bits can have been lost by the right shifting of the fractional part of the number with smaller exponent. The result is now normalized;* in the above example it is already normalized, but if D 27 and D 6, or D 27 and D −12, were added as floating point numbers, it would not be. (Why not?)

The result the user sees is now formed by discarding the low order word, and properly rounding the fractional part in the high order word.* Thus, bits 9 through 35 of the single word result contain the best possible approximation to the fractional part of the correct solution.

**We do not consider here the floating point instructions that perform arithmetical operations without normalization or without rounding of the result.*

The Floating point ADd and Round instruction FADR, and the Floating point SuBtract and Round instruction FSBR, are the counterparts of the arithmetical instructions ADD and SUB. Note that floating point instructions should be used only when both operands are already stored in normalized binary floating point form.

Modes

There is no difficulty with the action of the M and B modes in floating point instructions. As with the arithmetical instructions, there are two ways of conveying a numerical operand with the instruction itself. A literal may be used: just as we can write

```
ADD     AC,[1234567]
```

so we can write

```
FADR    AC,[1234567.0]
```

(Are these numbers equal?)

Alternatively, if the numerical operand will fit into a half word, we may use the Immediate mode, as in

```
ADDI    AC,12345
```

The Immediate mode of the floating point arithmetical instructions also accepts the result of the effective address calculation as the operand itself; but it interprets this as the *left* half of a floating point number (with zero right half). Consider for example an attempt to add D 2.0 to the contents of AC with

```
FADRI   AC,2.0
```

The floating point number D 2.0 assembles with zero sign bit; fractional part having 1 in bit 9 and zeros elsewhere; and with O 202 in the exponent field. As always with instructions in immediate mode, the right half word of this expression assembles into the right half of the instruction code word. But the right half word of the representation of D 2.0 consists of all zeros. So the given instruction assembles as

```
FADRI   AC,0
```

When this instruction is carried out, the quantity to be added into AC is calculated as follows: the effective address, which is 0, forms the left half of the word; the right half of the word is zero. Thus, the whole word is zero, and this represents the floating point number 0.

Suppose, however, we tried

```
FADRI   AC,202400
```

In this case, the floating point number to be added to the contents of AC is that represented by 202400,,0. But this is the normalized binary floating point representation of D 2.0. So this instruction would have the desired effect. However, it would be awkward to calculate the octal representation of every floating point number used in an immediate mode instruction. Observe that the operand we used was just the representation of D 2.0, shifted eighteen bits to the right. There is a special operator ← (this is _ on some terminals) that causes the assembler to shift the expression that follows. In our example, we could have

```
FADRI   AC,2.0←^D-18
```

The nature of the shift is that of a LSH instruction. Here the negative sign indicates a shift to the right.

Note that in immediate mode only nine bits are available for the fractional part of the number.

Exercises: (i) Which decimal numbers require no more than a nine bit fractional part for their exact representation in normalized binary floating point format?
(ii) How could MOVE AC,[2.0] be effected in one instruction, without using a literal?

If a floating point addition or subtraction instruction would produce an exponent greater than D 127, then OVERFLOW and FLOATING OVERFLOW are set in FLAGS. If a negative exponent of magnitude greater than D 128 would be produced, then OVERFLOW, FLOATING OVERFLOW, and FLOATING UNDERFLOW are set.

Multiplication and division of floating point numbers by floating point numbers are carried out by FMPR and FDVR. Both of these normalize and round the result, and can set OVERFLOW, FLOATING OVERFLOW, and FLOATING UNDERFLOW under the same circumstances as the other instructions. In addition, FDVR can set NO DIVIDE. In this case, the instruction is not carried out, and OVERFLOW and FLOATING OVERFLOW are also set. However, with normalized operands, this can only happen if an attempt is made to divide by zero.

Exercises: (i) Can the CAM- instructions be used to compare two floating point numbers? What about the CAI- instructions?
*(ii) Write a routine to determine whether a floating point number is actually an integer between -2^{35} and $2^{35} - 1$
(a) precisely;
(b) to within an approximation of D 0.000 001 either way, or thereabouts.

Conversion

It is often necessary to convert between fixed and floating point formats. The KI10 and KL10 processors have special instructions for this purpose. The FLoaT and Round instruction

```
FLTR    AC,MEM
```

produces from an integer in MEM a floating point number, which it places in AC (leaving MEM unaffected).

The FIX and Round instruction

```
FIXR    AC,MEM
```

creates in AC the integer closest in value to the floating point number in MEM. Note that a floating point number may be of too large a magnitude for this conversion to be meaningful; specifically, if the exponent in the floating point number is greater than thirty-five, the instruction will not be carried out, and OVERFLOW will be set.

FLTR and FIXR are not available on the KA10 processor. An integer can be converted to floating point form using the Floating SCale instruction

```
FSC     AC,X
```

which multiplies a floating point number in AC by 2^X. It does so by adding X to the exponent in bits 1 through 8. The result is normalized if it is not so already. Overflow or underflow may occur, and the flags are set accordingly. This instruction is the floating point counterpart of ASH.

Exercises: (i) FSC AC,233 will convert a positive integer less than 2^{27} contained in AC into normalized floating point format. Why?
(ii) Investigate the effect of FIXR when the source location contains a (positive or negative) floating point number whose fractional part is exactly 1/2. (FIXR is the ALGOL standard for real to integer conversion. The FORTRAN standard is provided by the FIX instruction.)
(iii) Write a routine for use with a KA10 processor to simulate the FIXR instruction.

(iv) What does the following routine do?

```
        MOVE    1,MEM
LAB:    MOVE    MEM
        FDVR    1
        FADRM   1
        FSC     1,-1
        JRST    LAB
```

How would you decide when sufficient accuracy had been obtained?

*(v) Write routines to evaluate various mathematical functions, such as square root, sine, and log. Write programs that use your routines.

FORTRAN Routines

Very efficient routines for evaluating arithmetical functions have already been written. In particular, the whole FORTRAN library of subroutines and functions is available to the MACRO-10 programmer. Conversely, a FORTRAN program may reference a MACRO-10 subroutine.

For the remainder of this section we assume a working knowledge of FORTRAN.

Suppose that, for example, we wish to use the FORTRAN function SQRT to evaluate the square root of a quantity. First we write a FORTRAN file, called, say, FSQRT.FOR, consisting of the following subroutine

```
SUBROUTINE FSQRT (X,Y)
Y=SQRT(X)
RETURN
```

Of course the name FSQRT is chosen for our convenience. The FORTRAN statement SUBROUTINE generates assembler language ENTRY and TITLE statements when the file is compiled. The assembler language code generated by the FORTRAN command RETURN ends with

```
JRA         16,(16)
```

so accumulator 16 must be used to call the subroutine with a JSA instruction.* It is a good idea to get used to reserving accumulator 16 for JSA — JRA calls for compatibility with FORTRAN subroutines. The parameters given in the SUBROUTINE statement are taken as the successive locations after the subroutine call. Thus, X will be picked up by an instruction such as MOVE @(16); and the result will in due course be passed back by MOVEM @1(16). Of course, all this is hidden within the output of the FORTRAN compiler, unless you choose to inspect it. It is a good idea to do so at least once.

Figure 4.5 illustrates the process of calling a FORTRAN subroutine from a MACRO-10 program.* First, not only the subroutine FSQRT but also the FORTRAN operating system program FORSE. must be declared external. (Note that in FORSE. the period is part of the name.) FORTRAN routines use accumulator 17 for pushdown list operations. Since there is no FORTRAN main program to set up the pushdown list, we must remember to do it. It is convenient for compatibility with FORTRAN programs to reserve accumulator 17 for this purpose in all programs requiring a pushdown list.

The subroutine is called by JSA 16,FSQRT.* As arguments, we must pass: first, the location in which the number whose square root is required is to be found—here it is MEM, which of course must be declared in the program; second, the location called ANS in which the square root is to be returned.

Arguments are passed by the ARG operator, which takes two parameters: the type of the argument, and the address of the argument, separated by a comma. The types of chief concern to us are: 0, which specifies integer; 2, which specifies real; and 3, which specifies logical. These

*See the note on FORTRAN compilers at the end of this section.

```
EXTERN  FORSE.,FSQRT
        ...
START:  MOVE    17,[IOWD  100,PSHLST]
        ...
        JSA     16,FSQRT
        ARG     2,MEM
        ARG     2,ANS
        ...                     ;return here with result in ANS
        ...
PSHLST: BLOCK   100
```

FIGURE 4.5 Calling a FORTRAN subroutine.

correspond to the word formats for arithmetical, floating point, and logical quantities. In Figure 4.5 we are specifying that MEM contains a floating point number whose square root is to be returned, as a floating point number, in ANS.

If we passed MEM by ARG MEM (equivalent to ARG 0,MEM), then MEM would be declared as containing an integer, and passed as such. The result would be similar to that of a FORTRAN statement Y=SQRT(I). Similarly, passing MEM as type 2, but ANS as type 0, is akin to the FORTRAN statement J=SQRT(X).

To execute the program TEST.MAC using the FORTRAN square root subroutine, simply

EX TEST, FSQRT ↵

We may collect such various subroutines as FSQRT into a library file using the FUDGE command and MAKLIB program, just as with assembler language subroutines. Or indeed, we can collect them into one library file together with our own assembler language subroutines. Since all subroutines must be compiled before insertion into the library, the work of FORSE. is already done, and there is no longer any need to declare it external in the calling program.

Exercises: (i) Compare the efficiency of the subroutines you wrote in Exercise (v) above with the corresponding FORTRAN subroutines. Use for this purpose the monitor call

RUNTIME AC,

which places in accumulator AC the *accumulated* run time of the job, in milliseconds. Accumulator AC must first be set to contain the job number (you are given this at LOGIN time); but if AC is set to zero, it is assumed by the monitor that the job issuing the call is meant.

(ii) Investigate how ARG is assembled. Why is the instruction JRA 16,(16) generated by the RETURN statement acceptable, rather than, in the case we discussed above, JRA 16,2(16) ?

(iii) Can the parameter specifying the type in an ARG statement affect the result of addressing the second parameter indirectly, as in MOVE @(16) ?

In Figure 4.6 we illustrate a trivial MACRO-10 subroutine. It can be called from a FORTRAN program, as long as it is declared to LINK-10 at execution time in the manner that is now familiar. In the FORTRAN program, there is no need to supply any EXTERN IADD statement, as the FORTRAN compiler will do this for us. The subroutine is called by a FORTRAN instruction such as

K=IADD(I,J)

The assembler language code generated by this instruction is

```
JSA     16,IADD
ARG     0,I
ARG     0,J
```

```
ENTRY   IADD

IADD:   0
        MOVE    @(16)
        ADD     @1(16)
        JRA     16,2(16)        ;return with result in AC0

        END
```

FIGURE 4.6 A MACRO-10 subroutine set up to be called from a FORTRAN program.

so the subroutine will properly pick up the values of the variables I and J. Note that a FORTRAN function always returns its result in accumulator 0, so this is where the subroutine must leave the value of K to be picked up.

Alternatively, the MACRO-10 subroutine could return the result to a third argument by including the instruction MOVEM @2(16) before the return. (Is it then *necessary* to change the return to JRA 16,3(16) ?) The FORTRAN program would then reference this subroutine by

```
CALL IADD(I,J,K)
```

with the same effect.

Exercise: What if we had named our MACRO-10 subroutine XADD rather than IADD?

NOTE

The above discussion assumes that FORTRAN programs are being compiled by the F40 compiler. If, however, the F10 compiler is used, some changes must be made. The F10 compiler calls subroutines with PUSHJ – POPJ, using a pushdown list in accumulator 17, rather than with JSA – JRA instructions. Accumulator 16 is also used to reference a block of locations at which are listed the arguments to be passed. The calling sequence is

```
MOVEI   16,ARGBLK
PUSHJ   17,SUBRTN
```

At ARGBLK the arguments are listed in just the same form as previously. You can check with your installation on the availability and functions of these FORTRAN compilers.

Exercise: What changes must be made in the subroutine of Figure 4.6 so that it can be called by a FORTRAN program compiled by the F10 compiler?

4.3 INPUT / OUTPUT

Until now, any information required by our programs has been typed in at the terminal at execution time, and the results of computations have been typed out at the terminal. In this section we consider how a program may obtain data from, and pass data to, a disk file.

To begin with, we shall write a program that will invite us to insert text, terminating with $; the program will then simply exit. Apparently nothing has happened, but examination of the directory will reveal the presence of a file, which, when typed, is found to contain the original text.

All Input / Output (I / O) is controlled by the monitor, in response to requests known as *monitor calls* issued from within a program. The first thing our program must do is have a *channel* opened between itself and the disk. This is a connection through which data may be passed between memory and disk. Sixteen channels are available to each user, numbered in the same way as accumulators. The request to open a channel is OPEN . Although this is a call to the monitor, it assembles in the standard instruction format. The chosen channel is specified as if it were an accumulator, and assembles into the accumulator field; this has no effect at all on the accumulator

bearing the same number. The effective address is that of a block of memory locations that must be set up by us to contain information required by this monitor call. So if we have, for ease of reading the program, defined CHAN=1 we proceed by

```
OPEN    CHAN,OPNBLK
```

where OPNBLK is the name we have chosen for the first location of the required block.

At OPNBLK we must reserve three words. In the first of these we must declare the *mode* of data transmission. Data transmission is always in bytes, and the mode specifies the size of byte that will be used (as well as other matters that will not concern us). If the first word at OPNBLK contains 0, then transmission will be in 7-bit bytes, packed five to each word; this is suitable for ASCII codes. If the first word at OPNBLK contains O 14, transmission will be in 36-bit bytes (this is D 36). A 36-bit byte is, of course, a whole word, so this mode is suitable for transmission of numerical data. When it comes to transmission, however, we must remain aware that we are dealing with bytes, and use the byte manipulation instructions we learned in Section 4.1.

In the second word, OPNBLK+1, there must appear a statement specifying the destination device. We shall deal here only with I / O to the disk; however, magnetic tape, DECtape, and other devices may be specified in a similar fashion. The statement for the disk is DSK. This statement is entered in a code called SIXBIT ; in SIXBIT, 6-bit codes are used for a limited range of characters, which includes numerals and uppercase letters. The format of a SIXBIT statement is the same as that of an ASCII statement.

The third word specifies whether we are going to perform input, output, or both, on the specified channel. The right half of this word is for input. In this program we shall not perform input, which is indicated by setting the right half of OPNBLK+2 to zero. We shall perform output; so in the left half of OPNBLK+2 we must put the address of the first word of yet another block, whose function we shall discuss below. We shall name this location OBUF . Thus, what we have so far is:

```
OPNBLK:  0
         SIXBIT   /DSK/
         XWD      OBUF,0
```

An XWD statement enters two half words in a single memory word; here it is an alternative notation to OBUF,,0.

The setup for I / O is illustrated diagrammatically in Figure 4.7. OPEN, OUTBUF, and INBUF are monitor calls. All the other symbols in the diagram are names of locations chosen by us for convenience of reference. So far, we have referenced with the OPEN call a three word block at OPNBLK, and stated there that for output we shall reference a block at OBUF.

As part of its response to the OPEN call, the monitor sets up the left half of location OBUF+1. We have called this location OBPTR because it will later be used as byte pointer. The monitor prepares it to handle bytes of the size we have specified via the contents of OPNBLK. The destination of the bytes is as yet undetermined, so the monitor can do nothing at this stage with the address part of location OBPTR.

For various reasons, monitor I / O requests might not be granted; as, for example, with an attempt to OPEN a channel to a busy or nonexistent device. The OPEN call deals with this contingency by causing the next line of the program to be skipped if the channel is successfully opened. So the instruction immediately following the OPEN call is performed only in case of failure: it is called the *error return*, with the next line following being the *normal return*. At the simplest level we could have

```
OPEN    CHAN,OPNBLK
EXIT                    ;if unable to open
...                     ;continue here
```

or EXIT could be replaced by a jump to an error routine.

The monitor does not transmit data in single bytes or words, but rather in *blocks*. The size of

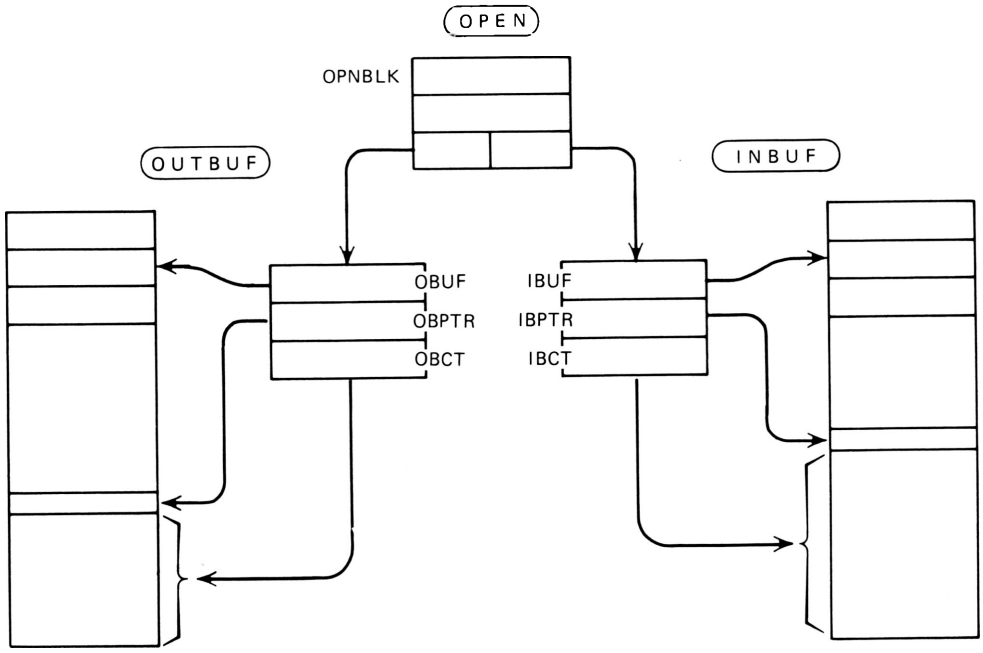


FIGURE 4.7 The I/O setup.

each block depends on the destination device. For the disk, each block contains O 200 words. The monitor uses an area of memory to hold data until a block is collected ready for output. Similarly, another area of memory is used to house a block of data on input. Such an area is called a *buffer*. The request to the monitor to set up a buffer for output on the channel called CHAN is

```
OUTBUF    CHAN,1
```

(the significance of 1 in the address field is discussed below). The monitor responds to this request by reserving a block of memory locations. It determines the size of the block in accordance with the physical requirements of the destination device; it discovers what the destination device is by referring to OPNBLK+1. For the disk, the buffer has to contain O 203 words, of which the first three are reserved for "bookkeeping" information.

The OUTBUF call also causes the monitor to put the address of the second word of the buffer into location OBUF. In addition, it sets bit 0 of OBUF to 1; this will later serve as an indication that output has not yet begun.

Exercise: Write a program to open a channel to the disk and set up a buffer for output. Check the effects of the monitor calls using DDT.

In order to perform output, we must specify the file in which our data will be written. The ENTER call causes the monitor to store a directory entry for later use. The calling sequence is

```
ENTER     CHAN,FILNAM
error return
normal return
```

Note that if the call is successful there is a skip, just as with OPEN.

FILNAM is the mnemonic chosen by us for the first word of a four word block that we must have set up for reference by the monitor when it performs the ENTER call. In FILNAM itself we put the name we want the file to have. In the second word we put the extension to the file name, without the period that constitutes the first character of extensions to file names. These must be

SIXBIT statements. The third and fourth words may be left null. The monitor returns information to this block, in a form that does not concern us here. We have chosen the name TEXT for our file, so our FILNAM block is

```
FILNAM:     SIXBIT      "TEXT"
            0
            0
            0
```

A complete program to write the file TEXT is in Figure 4.8. The first monitor call, RESET, is necessary to have certain technical initialization procedures performed. Reading through the program, we see that it responds to the $ character (chosen here to indicate end of text) with

```
            CLOSE       CHAN,
```

The monitor call CLOSE transmits any data currently in the buffer, adds to the directory the file name specified in the ENTER command, and closes the channel. Note that the comma after CHAN is necessary to ensure that the channel number assembles into the accumulator field, as the format of this call requires.

After CLOSE, the next instruction in sequence in the program is performed; there is no error return. The same is true of OUTBUF and RESET .

If any character other than $ is received, the program responds by depositing it into the output buffer; it uses for this purpose a byte pointer in location OBUF+1, which we have also called OBPTR. It keeps a count of the number of bytes remaining in the buffer in location OBUF+2, also known as OBCT.

```
        CHAN=1

START:  TITLE   IOTST1
        RESET
        OPEN    CHAN,OPNBLK
        JRST    ERROR
        OUTBUF  CHAN,1          ;one buffer
        ENTER   CHAN,FILNAM
        JRST    ERROR
        OUTSTR  [ASCIZ  /INSERT TEXT:  /        ]
L1:     INCHWL
        CAIE    33              ;all read?
        JRST    L2
        CLOSE   CHAN,
        EXIT
L2:     SOSG    OBCT            ;room in buffer?
        JRST    L4              ;no - get new one
L3:     IDPB    OBPTR
        JRST    L1
L4:     OUT     CHAN,
        JRST    L3

ERROR:  OUTSTR  [ASCIZ  'I/O ERROR'     ]
        EXIT

OPNBLK: 0
        SIXBIT  'DSK'
        XWD     OBUF,0

OBUF:   0
OBPTR:  0
OBCT:   0

FILNAM: SIXBIT  'TEXT'
        0
        0
        0

        END     START
```

FIGURE 4.8 A program to write a file.

The first time that line L2 of the program is reached, both OBCT and the right half of OBPTR still contain zero. So we immediately jump to line L4 and issue the monitor call OUT CHAN, whose calling sequence is

```
OUT        CHAN,
normal return
error return
```

The request OUT transmits the contents of the buffer to the destination device, then sets the whole buffer (apart from the bookkeeping words) to zero. This is called *writing* the file and *emptying* the buffer. It then sets up OBPTR to point to the byte preceding the first byte available for storage in the buffer; thus, the next byte to be deposited is correctly placed by an IDPB instruction. It also sets OBCT to contain the number of bytes available for data in the buffer.

However, if bit 0 of OBUF is set to 1, the OUT instruction will set it to zero, and set up OBPTR and OBCT as indicated above, but will perform no data transmission. Thus, there must be one "dummy" OUT monitor call to initialize the block at OBUF before output actually can begin.

Note that we have returned from the OUT call to line L3, rather than to L2. Thus, the first byte is deposited before the buffer byte count is first decreased. So when the count in OBCT reaches zero the buffer is full. If another byte were to be deposited, its destination would be beyond the end of the buffer; the OUT call would not transmit it, and it would be lost.

As mentioned above, there is no OUT for the final buffer of data, as CLOSE performs this function also; however, an extra OUT would do no harm. The monitor stores the final block with a count of the number of words of data in it, but any preceding blocks are regarded as full. So it is necessary to take care to fill buffers (except the final one) completely before transmission, otherwise what would appear as items of data equal to zero will be included as filler. This does not matter with ASCII text, but might cause trouble with numerical data.

Observe that the OUT monitor call has its following line as normal return, and skips as an error return. OUT and its input counterpart IN (discussed below) are the only monitor calls to skip as an error return. Note how we take care of the error return in our program.

Exercises:

(i) Suppose that the text THE is inserted. Just after the instruction at line L3 is performed for the second time, if the right half of OBUF contains 7036 what are the contents of locations:
   (a) the left half of OBUF;
   (b) OBPTR;
   (c) OBCT;
   (d) 0;
   (e) 7040.
   What is the address of the last word in the buffer?

(ii) Write a program that accepts text, but transmits numerals to one file and all other characters to another. You must perform all I / O operations for each file separately, giving each its own channel.

*(iii) Write a program that before accepting text will ask for the name of the file to be created. (The SIXBIT code for an uppercase letter is equal to its ASCII code minus O 40.)

Although we have not introduced it specifically, the effect of the SOSG instruction at line L2 should be obvious. Note that all the instructions

```
SOS-       AC,MEM
```

as well as

```
AOS-       AC,MEM
```

and

```
SKIP-        AC,MEM
```

will load the contents of the effective address into AC, unless AC is accumulator 0. This is the case whether there is a skip or not.

The block at OBUF is called the output *buffer ring header block*. In general, the call

```
OUTBUF       CHAN,n
```

where *n* is a positive integer, sets up *n* output buffers. In each buffer, the right half of the second word contains the address of the second word of the next buffer, with the last buffer referring back in this way to the first. This is why it is called a *ring* of buffers. Each OUT call empties whichever buffer is currently in use, and sets up the contents of the words in the buffer ring header block to reference the next buffer. Having more than one buffer can enable the I / O system to operate more efficiently, although very sophisticated techniques are required if much advantage is to be gained from having more than two. If the call OUTBUF CHAN, is given without specifying a number, the monitor will set up a ring of two buffers as a default. It is somewhat easier to keep track of what is going on if only one buffer is used, and we recommend that you restrict yourself in this way until you are more experienced with I / O.

Exercises:

(i) How could a program check whether the buffer ring referenced by the block at OBUF contains more than one buffer?

(ii) Write a program to create a file called NUMBRS, to consist of O 1000 locations containing successively the numbers 1 through O 1000. (You must set

```
OPNBLK: 14
```

for 36-bit bytes. Otherwise use the same I / O procedures as before.) How many disk blocks does NUMBRS take up? What if we had stored in it the numbers 0 through O 1000 ? Check this from your directory.

(iii) In ASCII mode (OPNBLK set to 0) files used as input to a FORTRAN program must be *line blocked*. For the meaning of this, define a line as a string of characters terminated by ↵ . Then a (computer) word within which a line ends must be filled out with zero bytes, and the new line started from the next computer word. Further, a line may not be split across output buffers. Write a program to write files in this fashion.

Reading a File

Now we shall write a program to read the file that we have just written. The actual effect of this program merely duplicates the command TYPE to the monitor. Once we can write a program to retrieve information from a file, however, the program can go on to make use of that information. Our next program should, therefore, be seen merely as illustrating the input process.

To read a file, just as to write one, a channel must be specified, and the OPEN call used to establish a correspondence between program and disk.

This time we shall be performing input; the third word of the block starting at OPNBLK must contain, in its right half, the location of the first word of the input buffer ring header block. In our program we have called it IBUF. At IBUF, the functions of the three locations parallel those of the output buffer ring header block of the program in Figure 4.8.

We set up a single buffer for input on our channel using the INBUF monitor call

```
INBUF        CHAN,1
```

Like OUTBUF , this call has no error return line.

Now we must request the monitor to seek the disk file that we wish to read. This is

accomplished by the LOOKUP monitor call.

```
LOOKUP    CHAN,FILNAM
error return
normal return
```

where FILNAM is the address of a four word block that must specify the file in exactly the same way as with the ENTER call. The error return line will, for example, be taken if there is no such file on the disk.

The monitor call IN is used to have data passed on the specified channel to the buffer: the buffer is *filled* and the file is *read.* As with OUT , the normal return for IN is the next line, with the error return being the second line after IN .

Unlike OUT , every occurrence of IN , including the first in a program, will fill a buffer with data—as long as there is data left in the file being read. If there is no more data left, IN will take the error return. Our program utilizes this by having as the error return line a jump to a suitable file closing routine.

The quantity of data read by an IN call is one disk block. Successive IN calls read successive blocks, beginning with the first block of data in the file. The monitor puts a copy of the data into the buffer, without in any way affecting the disk file.

We have arranged our program so that 1 is subtracted from the byte counter before a byte is loaded. Thus, after subtracting 1, there is one more byte left in the buffer than is indicated by the contents of IBCT. So a byte should be loaded at this stage, rather than calling for a new buffer of data, as long as the contents of IBCT are not less than zero. This is organized by the instruction

```
SOSGE     IBCT
```

If there is no skip, a buffer of new data is delivered.

You should compare this byte counting with what we did for output, and be sure that you fully understand how such differences as there are arise. Taking care of each single byte on input, as with output, is a matter of neither pedantry nor economy. It is simply that data may be lost if we try to write more into a buffer, or to read less out of it, than it holds.

The program to print out the file TEXT created by the previous program is in Figure 4.9.

Exercise: Write a program to print out the file NUMBRS that you created in Exercise (ii) above. Print out the numbers in octal notation, in eight columns. Check the number of blocks by having an extra line feed issued immediately before each IN call.

File Status Word

Our program to perform input contains a piece of very bad programming practice, which we shall now rectify. We took the error return of the IN call to indicate that the end of the file being read had been reached. Now indeed IN will skip under these circumstances; but other conditions can also result in an error return on an attempt to input data. For example, there could be something wrong with the disk itself. As things stand, we have no check on why the error return was taken, and so no sure knowledge that the file has been properly read. However, for each channel in use for I / O the monitor maintains a *file status word*, and the setting of certain bits in this word indicates the error conditions that have occurred. The only bit in which we are interested here is the *end-of-file bit*, which is bit 22 in the file status word; it is set to 1 when an IN call endeavors to read past the end of the file. The file status word is not a memory location available to the user; however, monitor calls are available to make any desired changes in the contents of the file status word. We shall consider here only certain calls by which bits in the file status word may be checked.

The call

```
STATO     CHAN,X
```

```
        CHAN=1

START:  TITLE   IOTST2
        RESET
        OPEN    CHAN,OPNBLK
        JRST    ERROR
        INBUF   CHAN,1
        LOOKUP  CHAN,FILNAM
        JRST    ERROR           ;if no such file
L1:     IN      CHAN,
        JRST    L2
        OUTSTR  EOF             ;error return - so end of file
        CLOSE   CHAN,
        EXIT
L2:     SOSGE   IBCT            ;room in buffer?
        JRST    L1              ;no - get new one
        ILDB    IBPTR
        OUTCHR
        JRST    L2

OPNBLK: 0
        SIXBIT  /DSK/
        XWD     0,IBUF

IBUF:   0
IBPTR:  0
IBCT:   0

FILNAM: SIXBIT  /TEXT/
        0
        0
        0

ERROR:  OUTSTR  [ASCIZ  'I/O ERROR'     ]
        EXIT

EOF:    ASCIZ   /
END OF FILE/

        END     START
```

FIGURE 4.9 A program to read a file.

will skip if any of the bits in the file status word for the channel CHAN that are masked by the right half word quantity X are set to 1. Bit 22 is masked by O 20 000, so the error return on IN should incorporate the routine

```
STATO   CHAN,20000
JRST    ERROR       ;not end-of-file
```

before continuing as before. Alternatively, the program could check that none of the other error bits are set to 1; these are bits 18 through 21 of the file status word. This check would be most easily accomplished using the STATZ monitor call, which causes a skip if all the masked bits are set to 0. (What is the mask in this case?)

Update Mode

Having learned how to create and read files, we next consider how to make changes in existing files. It is no use attempting this by performing an ENTER to a file already on the disk. The monitor will write a new file, containing the new data only. When a CLOSE is performed, the old version of the file will be deleted from the disk, with the new one taking its place; the file is *superseded.*

To amend an existing file, I / O must be performed in *update mode.* This is achieved by issuing the monitor calls LOOKUP and ENTER *in that order*, using the same channel and the same file name. Of course the channel must first be OPENed. Although LOOKUP and ENTER both reference four word blocks of identical format, the monitor amends the block when it carries out either of these calls. So after LOOKUP , the block is not suitable for use by ENTER . Either the

program can reset the block for ENTER , or, more simply, it can use a sequence such as

```
LOOKUP    CHAN, FILOOK
JRST      ERROR
ENTER     CHAN,FILENT
JRST      ERROR
...
```

where FILOOK and FILENT are the first words of separate four word blocks with identical contents.

The monitor maintains *pointers* indicating the block of the file being referenced, one for input and one for output, for each channel being used. Note that once LOOKUP and ENTER have been performed for a given file, both input and output may be effected for that file on the same channel; however, there must be two distinct buffer ring header blocks.

The LOOKUP call sets the input block pointer to 1, and the ENTER call sets the output block pointer to 1, in each case referencing the first block of data in the file. Every time an IN call is performed, the input block pointer is increased by 1, so the next following block will be read by the next IN call. This continues until there are no more blocks left, whereupon the end-of-file bit in the file status word is set. Performing an OUT call increases the output block pointer by 1, ready for the next OUT to write the next block of the file. However, if the program has passed no data to the output buffer, OUT will have no effect whatsoever. The monitor will not trouble to write an empty buffer. Note that a buffer containing items of data that happen all to be zero is not the same thing as an empty buffer. The monitor checks OBPTR to determine if any data has been passed. Depositing a null byte into the output buffer with an IDPB instruction, then performing an OUT , will write on the disk a whole block of zeros; only if this is the last block of the file will the monitor record the fact that all bytes except the first are "filler" and not meant to be part of the file.

Exercises:

(i) Write a program to create, in the simplest possible way, a disk file of O 20 blocks, in which successive blocks contain
  (a) successive integers, followed by O 177 words of zeros;
  (b) successive letters of the alphabet, as ASCII codes, followed by O 1177 zero bytes. (Why O 1177 ?)

(ii) Write programs to print out these files. Explain the results.

(iii) Write programs to access these files in update mode. Move any data you please to the buffer, then CLOSE the file. Now run again the programs to print out the files. What conclusions can you draw?

As you saw in Exercise (iii), in update mode the new data is written into the first block of the file. If an OUT is performed, the next buffer will be written into the second block, and so forth. Thus, we can change the contents of every block in our file, starting from the first. Note that the whole block is replaced by the new buffer contents, and that so far the only way we have of passing over a block without changing it is to retype its entire contents.

Exercises (continued):

(iv) Write a program to change all lowercase letters in a file to uppercase. Your program must go through the input buffer, amend bytes as necessary, and deposit them into the output buffer.

(v) Devise a method of storing partially filled blocks of numerical data, and printing out the resulting file. (Some data items may themselves be zero.)

User Block Control

The user's control over the input and output block pointers is by no means limited to advancing either of them one block at a time as a by-product of an IN or OUT call. With the User SET Input monitor call USETI , and the User SET Output call USETO , the pointers may be set to

any chosen block. Each of these calls must be assembled with the channel number in the accumulator field. The result of the effective address calculation is itself treated as the block number. Thus the call

```
USETI     CHAN,3
```

sets the input block pointer to reference the third block of data from the start of the file. The same effect can be achieved, if accumulator N contains the quantity 3, by issuing the call

```
USETI     CHAN,(N)
```

If the block number is contained in a memory location, indirect addressing will be needed. Observe that no input is performed by this call; it merely sets the pointer ready for a subsequent IN call to read the desired block. Note also that the output block pointer is unaffected.

Using these calls, any block of a file may be accessed as easily, and as quickly, as any other. When there is more than one buffer, the monitor will fill as many as possible on input, and empty as many as possible on output. So to keep track of the block being referenced, a program using the USETI monitor call should have only one buffer for input; and likewise for output if USETO is being used.

Exercises: (i) Write a program to print out the fourth block of NUMBRS.
*(ii) Write a program to interchange the contents of the *m*th and *n*th blocks of a given file, where *m* and *n* are integers typed in at the terminal in response to requests issued by the program. (Keep careful track of both block pointers.)

Neither USETI nor USETO will necessarily give any spontaneous error indication if a block that is not part of the file is specified. If a USETI specifies a block number larger than that of the last block of the file, the end-of-file bit in the file status word is set, and an IN call will now fail. (If USETI to an existing block is subsequently issued, the monitor will clear the end-of-file bit, and input will again be possible.) This treatment of too large block numbers is in effect as long as the block number specified is not so large as to look like a negative number of small magnitude (in 18-bit twos complement form). For example, USETI CHAN,777 770 assembles in the same way as USETI CHAN,−10. And the calls USETI CHAN,−2 through to USETI CHAN,−10 have a special function; they set the input block pointer to the second through eighth block of the O 10 blocks of information about the file, which the monitor stores with it. These blocks are known collectively as the *retrieval information block*, or RIB. The first block of the RIB is, regrettably, called the prime RIB. The call to set the input block pointer to the prime RIB is out of sequence: it is USETI CHAN,0. The call

```
USETI     CHAN,−1
```

is rather special. It sets the end-of-file bit, thus inhibiting further input. But, unlike USETI for any other block number, it also affects the *output* block pointer; it sets this to reference the block after the last block in the file. Thus, in update mode, this call enables data transmitted by subsequent OUT calls to be appended to the file. If the file contains X blocks, then USETO CHAN,X+1 has the same effect on the output block pointer; but of course the program has first to determine X, which takes time. Note that the count of data items in what was formerly the last block of the file will be lost.

A program might, when writing a file, keep in the first word of each block a count of the number of bytes of data in that block. On a later update, after input a block can be set up for reading by

```
AOS       IBPTR
MOVE      1,@IBPTR
MOVEM     1,IBCT
```

or, more elegantly, by

```
AOS     1,IBPTR
MOVE    1,(1)
MOVEM   1,IBCT
```

Exercises: (i) Write a routine to output data in blocks, storing the byte count in the first word of the block.

*(ii) Write a program to update a given file, which will request the value of an integer *n*, then insert input text after the *n*th block of the file. If *n* = 0, the text is to be inserted at the beginning of the file. If the number of blocks is no greater than *n*, the text is to be appended to the file.

If the call USETO CHAN,X is issued, with X greater than the number of blocks in the file, the monitor will allocate any intervening blocks as part of the file. So if a file contains O 10 blocks, the call USETO CHAN,70 followed by OUT CHAN, will create a file of O 70 blocks, with the new text being written in the last one, and blocks numbered O 11 through O 67 empty. A really large value for X will exceed the user's memory allotment; the monitor will stop the program and print an error message. The call USETO CHAN,0 will merely set bit 21 in the file status word; this inhibits further output.

The call

```
USETO   CHAN,-1
```

has a special effect. The output block pointer is set to the block on which I / O was most recently performed; that is, the block used in the most recent IN or OUT call. So a program can update a block of a file with the sequence

```
IN      CHAN,
USETO   CHAN,-1
```

followed by the necessary emendations and an OUT .

A convenient way to move buffers of data around in memory is provided by the BLock Transfer instruction

```
BLT     AC,MEM
```

This is a general instruction to move a block of words. It will start by moving the contents of the location addressed by AC left to the location addressed by AC right. It will continue moving successive words, until a word is moved to the location given by the effective address calculation. Thus, the contents of the input buffer may be moved to locations LOC+1 through LOC+200 by

```
HRLZ    1,IBPTR
HRR     1,LOC
BLT     1,LOC+200
```

(What will LOC contain after this?) The contents of the source block—in this case the input buffer—are unaffected.

For technical reasons, BLT should not use the same accumulator AC to index the address; nor should it be assumed that the contents of AC are left unchanged by this instruction. Note that BLT moves its first word before checking whether enough words have been moved; so, whatever the effective address, BLT always moves at least one word.

Exercises: (i) Write a program to read a file in which the first block contains not data, but information specifying the order in which the remaining blocks are to be read. (One way of holding this information would have successive words of the form *m*,,*n*

to indicate that the *n*th block is to be read immediately after the *m*th. A single entry of the form −1,,*k* could indicate that the *k*th block is to be read first.)

(ii) Create a file in this form for your program to read.

(iii) Write a program to update this file by always "physically" appending any additional data, but amending the first block according to where the data should "logically" be inserted.

(v) Write a program to accept a string of characters typed in at the terminal, and search a given text file to see whether the string is to be found. Your program should work even if the string straddles two or more blocks of the file.

(vi) Improve the program of the last exercise so that if the search is successful a new text string may be substituted for the first.

(vii) Carefully study and annotate the program of Figure 4.10. It is a simplified version of the program used to prepare the index for this book.

4.4 MONITOR ASSISTANCE

In this section we collect together a variety of procedures that are all put into effect by the monitor's intervention in the course of executing a program.

Let us first look a little more closely at how this intervention comes about. Many of the operations we have met, for example, OUTCHR, OPEN, RUNTIME, and EXIT, assemble as codes that actually have no meaning whatsoever as far as the machine hardware is concerned. Now whenever, while executing a program, the central processor encounters undefined code, the monitor is invoked. The code is said to *trap* to the monitor. Traps to the monitor occur under various other circumstances, such as pushdown list overflow, illegal memory reference, or ^C typed at the terminal. The action taken by the monitor depends on the cause of the trap.

Suppose that the central processor encounters

```
INBUF   CHAN,1
```

in the form of assembled binary code. First the effective address, equal to 1 in this case, will be calculated and stored. The central processor now finds that the operation code contained in bits 0 through 8 (it is actually O 64) has no meaning to it as an instruction; so it calls on the monitor. To the monitor the code is indeed familiar; it specifies a routine to achieve the result with which we are familar. All operation codes between O 40 and O 100 are of this type. They are called *Monitor Unimplemented User Operations,* abbreviated to Monitor UUO's, or to MUUO's. The description of them as *unimplemented* refers to their being not implemented by the *hardware,* but rather as invoking software routines. The usual effective address calculation is carried out for an MUUO, although the result is in many cases not interpreted as an address. For OPEN, ENTER, and LOOKUP it is an address. For STATO and STATZ it is a mask; for USETI and USETO a block number; for INBUF and OUTBUF the number of buffers. The accumulator field may specify something other than an accumulator; with the I / O MUUO's it specifies a channel, and the use of the channel has no affect on the accumulator bearing the same number.

Two of the MUUO's are of particular importance because each of them generates a whole class of operations. Operation code O 051 is TTCALL . For this code the accumulator field specifies a function code, by which the monitor is informed of which operation in the class is wanted. The assembler recognizes special mnemonics for the operations resulting from different accumulator fields. For example, TTCALL 1,MEM is OUTCHR MEM; TTCALL 3,MEM is OUTSTR MEM; and TTCALL 4,MEM is INCHWL MEM. It follows that OUTCHR and so forth cannot specify an accumulator field. These mnemonics for the TTCALL code (and those for CALLI discussed below) are in a symbol table that the assembler searches last of all tables. When found, they are put by the assembler in the table it draws up of symbols defined by the user, and appear as such in a program listing.

```
;MACRO to set up byte pointer in BP
;rh of X contains number of 7-bit bytes
;starting from first byte in WRD
;AC usage: 0, 1


DEFINE  SETUP   (BP,X)
<       JSA     16,$SETUP
        ARG     BP
        ARG     X       >

DEFINE  READ
<       JSA     16,$READ
        STATZ   CHAN,740000
        JRST    IOERR
        SETZM   LIST(N)
        OUTSTR  [ASCIZ  /FILE READ
/       ]       >

DEFINE  WRITE
<       JSA     16,$WRITE
        OUT     CHAN,
        JRST    .+2
        JRST    IOERR
        CLOSE   CHAN,
        OUTSTR  [ASCIZ  /FILE WRITTEN
/       ]       >

        BP1=2
        BP2=3
        CH1=4
        CH2=5
        CT1=6
        CT2=7
        N=10
        CHAN=1

$SETUP: 0
        HRRZ    @1(16)
        SOS             ;byte numbering starts at
        IDIVI   5
        ADDI    WRD     ;rh set up in AC 0
        IMULI   1,7
        SUBI    1,44    ;^D 35
        IMULI   1,-10000        ;for bits 0 - 5
        ADDI    1,700           ;for bits 6 - 11
        HRLZM   1,@(16)
        HRRM    @(16)
        JRA     16,2(16)

$READ:  0
        SETZB   CT1,CT2
        SETZ    N,
        MOVEI   1
        MOVEM   LIST
        MOVE    BP1,[POINT 7,WRD]
        RESET
        OPEN    CHAN,OPNBLK
        JRST    IOERR
        INBUF   CHAN,1
        LOOKUP  CHAN,FILNAM
        JRST    IOERR
R1:     IN      CHAN,
        JRST    .+2
        JRA     16,(16)
R2:     SOSGE   IBCT
        JRST    R1
        ILDB    IBPTR
        CAIL    40
        JRST    R3
        CAMN    CT1,CT2
        JRST    R2
        HRRZM   CT2,LIST+1(N)
        AOS     CT1,LIST+1(N)
        SUB     CT1,LIST(N)
        HRLM    CT1,LIST(N)
        MOVE    CT1,CT2
        AOJA    N,R2
R3:     IDPB    BP1
        AOJA    CT2,R2

OPNBLK: 0
        SIXBIT  "DSK"
        XWD     OBUF,IBUF

OBUF:   0
OBPTR:  0
OBCT:   0

IBUF:   0
IBPTR:  0
IBCT:   0

FILNAM: SIXBIT  "WORDS"
        0
        0
        0

FILBLK: SIXBIT  "WORDS"
        0
        0
        0

$WRITE: 0
        OUTBUF  CHAN,1
        SETZ    N,
        ENTER   CHAN,FILBLK
        JRST    IOERR
WR1:    SKIPN   LIST(N)
        JRA     16,(16)
        SETUP   BP1,LIST(N)
        HLRZ    CT1,LIST(N)
        MOVEI   12
        JRST    .+2
WR2:    ILDB    BP1
        SOSG    OBCT
        OUT     CHAN,
        JRST    .+2
        JRST    IOERR
        IDPB    OBPTR
        SOJGE   CT1,WR2
        CAIN    15
        AOJA    N,WR1
        MOVEI   15
        JRST    WR2+1


START:  READ
        SETZM   N
        MOVE    BP1,[POINT 7,WRD]   ;BP1 set up
        MOVEM   BP1,BP2
        HLRZ    LIST
        IBP     BP2
        SOJG    .-1                 ;BP2 set up

NEXT:   HLRZ    CT2,LIST+1(N)
        JUMPN   CT2,CHECK-1
        OUTSTR  [ASCIZ  /FILE SORTED
/       ]
        WRITE
        EXIT
        HLRZ    CT1,LIST(N)

CHECK:  SOJL    CT1,NOSWAP
        SOJL    CT2,SWAP
        ILDB    CH1,BP1
        ILDB    CH2,BP2
        CAILE   CH1,140
        SUBI    CH1,40
        CAILE   CH2,140
        SUBI    CH2,40
        CAMGE   CH1,CH2
        JRST    NOSWAP
        CAMN    CH1,CH2
        JRST    CHECK

SWAP:   MOVE    LIST(N)
        EXCH    LIST+1(N)
        MOVEM   LIST(N)
        JUMPE   N,.+2
        SOJA    N,.+2

NOSWAP: AOS     N
        SETUP   BP1,LIST(N)
        SETUP   BP2,LIST+1(N)
        JRST    NEXT

IOERR:  OUTSTR  [ASCIZ  "I/O ERROR"     ]
        EXIT

WRD:    BLOCK   4000
LIST:   BLOCK   1000

        END     START
```

FIGURE 4.10 A program to prepare an index.

Operation code O 47 is CALLI . With this code, the accumulator field usually specifies an accumulator; however, the effective address calculation specifies not an address, but a function code. For example, CALLI 12 is EXIT; CALLI 0 is RESET; and CALLI AC,27 is the RUNTIME AC, monitor call.

Terminal Control

CALLI AC,116 is the

```
TRMOP.    AC,
```

monitor call. Note that the last character of this call's mnemonic is a period. This is the case with all of the more recently introduced CALLI calls (those with address O 110 and above). The function of TRMOP. is to enable the user to control various characteristics of the terminal. Many of these can be effected by SET TTY monitor commands before running a program. However, it is useful for a program to be able to set the terminal as it requires. This is particularly true for systems programs. For example, the LOGIN program uses this monitor call to suppress echo of your password.

Using the TRMOP. monitor call is a several stage process. With this call the monitor must be supplied with a means of identifying the terminal you are referring to, in the form of a number called the *universal device index* (UDX) of the terminal.

The universal device index is supplied by the monitor in response to the CALLI AC,115 call, for which the mnemonic

```
TRMNO.    AC,
```

is accepted. The monitor will set AC to contain the UDX. However, in order to do so, the monitor must be told which terminal's UDX is wanted! Every user is perfectly free to obtain the UDX of every terminal connected to the system at which a job is running. The number of the job is the information which the monitor requires to carry out this call, and it must be placed in AC before the call is issued. The actual job number must be supplied; the monitor will not assume, as it does with the RUNTIME call, that if AC contains zero the user's own job is meant. So the calling sequence is

```
move job number into AC
TRMNO.    AC,
error return
normal return
```

The error return would, for example, be taken if no job with the number supplied exists.

The job number is supplied by the monitor when you LOGIN, and you can receive a reminder of it at any time when you are in communication with the monitor by giving the command

PJOB ↵

However, a program should not have to request the user to input the job number. A program can obtain the number of the job under which it is running by issuing a monitor call. The mnemonic for this call, which is CALLI AC,30 is the same as that of the monitor command to obtain the job number.

Let us examine this in sequence. First we read the job number of our own job into AC by

```
PJOB      AC,
```

Next we replace the job number in AC with the UDX of the terminal to which our job is attached, by

```
TRMNO.    AC,
JRST      ERROR
```

Now we are ready for the TRMOP. monitor call. It can be used to obtain information about a setting of the terminal, or (limited by the privileges of the user and the capabilities of the terminal) to change such a setting. The calling sequence is

```
set up AC
TRMOP.    AC,
error return
normal return
```

Unfortunately, AC has *not* already been set up by the TRMNO. monitor call; although TRMOP. requires the UDX, it does not want it in the accumulator it references. It must instead find in the right half of AC the address of the first word of a block that is used for passing parameters with this call. The number of words in the block varies with the use to which TRMOP. is being put. The monitor must find the number of words in the block specified in the left half of AC when it comes to carry out this call.

When TRMOP. is used to check a characteristic of the terminal, a two word block is required. Thus, the call is preceded in this case by an instruction such as

```
MOVE      AC,[2,,ADR]
```

Of course before this is done, the UDX must be put where the monitor will want to find it; this is in the second word of the two word block.

The first word in the block must be set up to contain the code for whichever TRMOP. function is required. The function code for checking a characteristic is always a four digit octal code whose first two digits are 10. As an example, suppose we have a program that will display its results graphically if the terminal has display capabilities, and otherwise just print the results out in columns. The code for checking the display capabilities of the terminal with the TRMOP. call is 1016. A routine for carrying out this check is in Figure 4.11.

The monitor will set AC to indicate the result of a check carried out in response to the TRMOP. call. In this case, it will set AC to contain 1 if the terminal is a display device, 0 if it is not.

Exercises: (i) The carriage width is the maximum number of characters that a typed line may contain. The monitor will return it in AC in response to a TRMOP. call with a function code of 1012. Write a program that will supply line numbers, right justified as far to the right on the page as possible, for any file (which you must LOOKUP and ENTER). Restrict printout of the numbers to lines that have enough room at the right end.

(ii) In response to a TRMOP. call with function code 1003, the monitor will set AC to contain 0 if the terminal is set to print lowercase as well as uppercase letters, 1 if it will print uppercase only. Write a routine that will convert lowercase to uppercase letters when the terminal is not set for lowercase.

```
FJOB    AC,
TRMNO.  AC,
JRST    ERR
MOVEM   AC,ADR+1
MOVE    AC,[2,,ADR]
TRMOP.  AC,
JRST    ERR
...
...
ADR:    1016
        0
```

FIGURE 4.11 Routine to check the display capabilities of the terminal.

```
        AC=1

START:  TITLE   LC
        PJOB    AC,
        TRMNO.  AC,
        JRST    ERR
        MOVEM   AC,ADR+1
        MOVE    AC,[3,,ADR]
        TRMOP.  AC,
ERR:    OUTSTR  [ASCIZ  /ERROR!/        ]
        EXIT

ADR:    2003    ;code for uc/lc
        0
        0       ;indicates lc capabilities

        END     START
```

FIGURE 4.12 A program to set the terminal to lowercase.

When the TRMOP. call is used to set a characteristic of the terminal, a three word block is required. So before the call is issued, an instruction such as

```
MOVE        AC,[3,,ADR]
```

is required. As before, the UDX should previously be moved into the second word of the block. The first word of the block again contains the function code. The code to set a given characteristic is always O 1000 more than the code to check it; so it will be a four digit octal number whose first two digits are 20. The third word of the block must give the value to which the characteristic is to be set. With characteristics taking numerical values (such as carriage width) the desired value is given. With characteristics that are either present or not (such as lower case capabilities) this word must contain 0 or 1. In all cases, the value to be passed is the same as the response the monitor would give on a checking TRMOP. call, if the terminal were set as desired.

Figure 4.12 is a program that has the same effect as the SET TTY LC monitor command. As indicated in Exercise (ii) above, 0 specifies lowercase capabilities; the third word of the block is set accordingly.

Exercises: (i) Amend your program that supplies line numbers for a file by enlarging the carriage width sufficiently to provide space for the numbers, if possible.

(ii) Function code 2007 controls whether characters you type in will appear at the terminal (be echoed). If the third word of the block referenced by the TRMOP. call contains 0, they will be echoed; if 1, they will not. Write a program that mimics the LOGIN procedure.

Traps and Intercepts

We conclude with a discussion of monitor intervention that is not the normal direct consequence of performing an instruction of the program. This occurs when certain errors result, or when a ^C is transmitted. Some errors will always cause the monitor to stop the program and print an error message. There are, however, several cicumstances under which the monitor will first check whether the program itself gives directions as to what should be done; only if no alternative has been provided will it then stop the program.

To carry out its check the monitor examines the user's *Job Data Area*. This consists of locations 0 20 through O 140 of the user's memory. The Job Data Area stores information relating to the program. Some locations in it are set by the monitor and may be of interest to the user; others are set by the user and utilized by the monitor. Each of these locations has a six character mnemonic code, of which .JB are the first *three* characters. Since the assembler must be activated to search a symbol table for these mnemonics, they should be introduced in a program as external symbols.

Suppose that pushdown list overflow has occurred. The monitor will first check whether the user

has stipulated that a routine will be available under these circumstances (*enabled* a trap). Traps are enabled by a special monitor call, discussed below.

If a trap has been enabled for this condition, the monitor first takes care that operations can later be resumed where they left off. It does this by storing the contents of PC in location .JBTPC in the Job Data Area. The monitor then transfers control back to the user's program, starting from whatever instruction it finds in location .JBAPR in the Job Data Area.

Location .JBAPR will not contain an instruction unless the program itself has already put one there. It must be a jump instruction to a routine for handling the condition that caused the trap (the *trap servicing routine*). Location .JBAPR must be set up before issuing the monitor call that enables the trap.

Traps are enabled by the CALLI AC,16 monitor call, recognized by the mnemonic

```
APRENB    AC,
```

The contents of AC tell the monitor which traps should be enabled. Each available trap is represented by a certain bit in the right half of AC; the monitor will enable those traps whose bits are set to 1. The bit corresponding to pushdown list overflow is bit 19. This bit is masked by O 200 000. So to enable this trap only, we would have the instruction

```
MOVEI     AC,200000
```

before the APRENB call. When issued in this form, the APRENB call enables a trap for one occurrence of the respective condition only. The trap servicing routine would then have to deliver another APRENB call (with bit 19 set in the AC) to re-enable the trap, before attempting to continue operations. However, setting bit 18 (which is masked by O 400 000) in AC enables the traps set by the other bits indefinitely, regardless of how many times the conditions for which traps are set occur. This is done in Figure 4.13, which is a program illustrating how to handle pushdown list overflow.

Our trap servicing routine in Figure 4.13 is very simpleminded, and meant merely to serve as an illustration. If the pushdown list is to be limited to a predetermined size, this might as well be set as the limit to begin with; no attempt should then be made to exceed this preset limit. A more valid use of overflow trapping occurs when the length of the list is not predetermined; in particular, when the user wishes to take up all the core that the system can provide.

The highest memory address available to the user's program is put by the monitor into location .JBREL in the Job Data Area. A program might so limit the pushdown list that overflow occurs just when all available core is used up. The trap servicing routine can then request more core (we show how to do this below). If the request is granted, the limit on the pushdown list can then be increased to match the new contents of .JBREL.

If an address beyond that given by the contents of .JBREL is referenced by a program, the monitor will stop the program, and print

?ILL MEM REF AT USER PC *number*

However, illegal memory reference can also be trapped. This is done by an APRENB call with bit 22 set in AC; note that bit 22 is masked by O 20 000.

The contents of .JBREL can be changed only by the monitor. More core cannot be obtained by meddling with this location. Instead, the monitor call CALLI AC,11 must be used; this is

```
CORE      AC,
```

The program must first set up AC to contain the highest desired address. This can simply be one more than the current contents of .JBREL

```
MOVE      AC,.JBREL##
AOJA      AC,.+1
CORE      AC,
error return
normal return
```

```
        AC=1
        N=2
        P=3
        CT=4
        T=5

START:  TITLE   PSHTRP
        MOVE    P,[IOWD  3,MEM]  ;pushdown pointer
        MOVE    [PUSHJ  P,PDLOV]  ;set up trap
        MOVEM   .JBAPR##          ;servicing routine
        MOVEI   AC,600000        ;pushdown list, repetitive
        APRENB  AC,              ;set trap
        SETZB   CT               ;initialize
        MOVEM   AC
        AOS
        PUSHJ   P,S1
        JRST    .-3

S1:     IDIVI   AC,10            ;octal print out
        HRLM    N,(P)
        JUMPE   AC,.+3
        PUSHJ   P,S1
        SKIPA
        PUSHJ   P,S2             ;to format routine
        HLRZ    N,(P)
        ADDI    N,60
        OUTCHR  N
        POPJ    P,

S2:     SOJG    CT,.+4           ;column count
        OUTCHR  [15]
        OUTCHR  [12]
        MOVEI   CT,10
        MOVE    T,P              ;spacing routine
        OUTCHR  [40]
        AOBJN   T,.-1
        POPJ    P,

PDLOV:  PUSHJ   P,LIMIT          ;check if more allowed
        SUB     P,[1,,0]         ;yes - increase limit
        JRSTF   @.JBTPC##        ;continue

LIMIT:  CAIL    1000             ;limit size of list
        EXIT
        POPJ    P,

MEM:    BLOCK   10

        END     START
```

FIGURE 4.13 A program to trap pushdown list overflow.

Asking for only one more location is adequate because the monitor will in fact supply rather more. Core is always allocated in units of O 2000 words (2K) on the KA10, O 1000 words (1K) on the KI10 and KL10 processors. A program should be sure not to waste time by requesting core when it is not needed; nor should it waste space by requesting core beyond its immediate needs.

The error return will be taken if no more core is available. The program must then do its best with the core it already has.

Exercises: (i) What is the effect of line PDLOV+2 in the program of Figure 4.13?

(ii) Look back at the programs you wrote to deal with arithmetic overflow (Section 3.3). Instead of having to check frequently for this condition, amend your programs by enabling a trap for it. (Arithmetic overflow is enabled by APRENB AC, with bit 32 of AC set to 1. Bit 18 has the same function as before.)

(iii) Do likewise for floating point overflow (Section 4.2). This corresponds to bit 29 in AC.

```
;this program illustrates ^C intercept
;2 ^C's will stop the simple background job
;and the program will wait for input
;any character will let the program continue
;from where it stopped without re-enabling
;except for ^A which will stop the job
;or ^X which will continue and re-enable
;or ^B which will re-start and re-enable

        AC=1
        N=2
        P=3
        CT=4
        T=5
        F=6

START:  TITLE   INTCPT
        MOVEI   INTBLK          ;set up for
        MOVEM   .JBINT##        ;intercept
        MOVE    P,[IOWD  10,MEM]
        SETZB   CT              ;start background job
        MOVEM   AC
        AOS
        PUSHJ   P,S1
        JRST    .-3

S1:     IDIVI   AC,10           ;octal print out
        HRLM    N,(P)
        JUMPE   AC,.+3
        PUSHJ   P,S1
        SKIPA
        PUSHJ   P,S2
        HLRZ    N,(P)
        ADDI    N,60
        OUTCHR  N
        POPJ    P,

S2:     SOJG    CT,.+4
        OUTCHR  [15]
        OUTCHR  [12]
        MOVEI   CT,10
        MOVE    T,P
        OUTCHR  [40]
        AOBJN   T,.-1
        POPJ    P,

INTBLK: XWD     4,INTLOC        ;4 words,,starting place
        XWD     0,2             ;no message,,^C intercept
        0               ;Monitor puts last PC here
        0               ;Monitor puts class bit in LH

INTLOC: HLRZ    F,INTBLK+3      ;check why interrupt
        CAIE    F,2             ;bit 34 for ^C
        EXIT                    ;exit if not
        INCHRW  F               ;get user input
        CAIN    F,1             ;exit if ^A
        EXIT
        CAIN    F,2             ;restart and
        JRST    I1              ;re-enable if ^B
        PUSH    P,INTBLK+2      ;save last PC
        CAIN    F,30
        SETZM   INTBLK+2        ;re-enable if ^X
        POPJ    P,              ;continue

I1:     SETZM   INTBLK+2        ;re-enable
        JRST    START+2         ;note addr for restart!

MEM:    BLOCK   10

        END     START
```

FIGURE 4.14 A program to intercept ^C.

Our final topic is ^C. Preventing ^C (or two ^C's if calculation is in progress) from immediately stopping the program is called *intercepting* ^C. We recommend that you experiment with ^C intercept only when an operator is on duty at your installation, just in case you make an error that leaves you with no way of exiting from a program.

When a user types ^C at the terminal, the monitor immediately examines location .JBINT in the Job Data Area. If this contains zero (which will be the case unless the program has put something there), the monitor will stop the program. Otherwise, the monitor takes the contents of .JBINT to be the address of the first word of the intercept block. In the illustrative program of Figure 4.14 we have set up location .JBINT to contain the address of the location we have called INTBLK.

At INTBLK a block of three or four words must be provided. If there are four words in the block, the monitor will return information in the fourth. The first word in the block must be set up to contain in its left half the number of words in the block. The right half of this word should contain the address of the next instruction the user wishes to be carried out when an intercept occurs.

In the second word, if bit 0 is set to 1, any error message available will be printed at the terminal when an intercept occurs. Our program does not trouble to set this bit, since there are no error messages associated with ^C intercept. The right half of this word must specify the conditions for which an intercept is desired: bit 34 is set to 1 to specify ^C.

If the monitor finds that the third word of the block (INTBLK+2 here) does not contain zero, it will stop the program in any case. Otherwise it will store the contents of PC there. (How does our program use this?) So if the intercept is to be re-enabled, INTBLK+2 must be cleared before leaving the intercept routine.

If a four word intercept block has been specified, the monitor will set in the *left* half of the fourth word the same bit which the user sets in the *right* half of INTBLK+1 to enable the intercept that has occurred. Our program checks that ^C was the cause of the interruption; this check is just for illustration, as in fact we enabled for nothing else.

Exercises:

(i) Where, and why, in Figure 4.14 is information that has been pushed down with a PUSH popped up with a POPJ?

(ii) What would be the effect of another ^C being received just before performance of the POPJ P, instruction at the end of routine INTLOC?

(iii) Amend your I / O programs of Section 4.3 so that on receipt of a ^C all open channels are closed before exiting. The main use for ^C intercept is to ensure that work already done is not lost.

(iv) Trying to write a large file may result in your disk quota being exceeded. This condition may be intercepted by setting bit 31 in INTBLK+1. Write a program to handle this situation by closing the file when it reaches the maximum permitted size.

APPENDIX A

DDT

DDT is the Dynamic Debugging Technique of the DECsystem-10. To use DDT on your program called, say, TEST.MAC, instead of EXecuting it, DEBug it:

```
DEB TEST ↵
```

The monitor will respond with

```
MACRO:    .MAIN
LINK:     Loading
[LNKDEB DDT Execution]
```

followed possibly by warnings of potential errors. However, if you begin using DDT with programs that run (such as those in this book), there will be no warnings.

You can check some of the facilities of DDT even before you let it execute your program. Suppose the line

```
PRINT:    IDIVI    N,10
```

appears in your program. The memory location in which the instruction IDIVI N,10 is stored is recognized by the name PRINT. You can see the contents of this location by typing

```
PRINT/
```

(do not follow this with a ↵), whereupon DDT will respond, on the same line, with

```
IDIVI N,10
```

The symbol / refers DDT to the word whose name has just been typed, and instructs it to type out the contents of the word. It is important to realize that DDT has done two things before typing out the contents of the location: it has set its location indicator, or *pointer*, to reference location PRINT; and it has *opened* location PRINT.

Once a location has been opened, its contents may be changed. Suppose you want to change the 10 to 12 in location PRINT. Then, after DDT has typed out IDIVI N,10 you type in the entire

new contents for PRINT; that is, you type

```
IDIVI N,12
```

and this time you do follow it with a ↵ .

DDT reads what you type at the terminal without waiting for a ↵ ; characters are transmitted as soon as you type them (INCHRW rather than INCHWL). This is already apparent from the effect of typing / after the name of a location. Now pressing the TAB key, which we shall denote by →| , issues a particular instruction to DDT. So when you type IDIVI N,12 ↵ to change the contents of PRINT, you must use a space as separator, and not →| .

You can check that DDT has done as you wanted by again typing

```
PRINT/
```

Since DDT accepts characters immediately, the ↵ after the new contents specified for location PRINT was not necessary just to transmit the characters. In fact, it instructs DDT to *close* the word, with the new contents. Once you are happy with the contents of PRINT as typed out by DDT, just type in a ↵ . If no new contents have been specified, DDT will close the word, leaving the contents unchanged.

It might happen that, while you are typing the new contents of a word, you reach the far right of the paper or screen. Just keep typing! The carriage will automatically return when the end of the line is reached, without affecting DDT. However, if you type a ↵ , it will instruct DDT to close the word, perhaps prematurely.

The changes you make in your program while using DDT are not preserved in your file; they are lost as soon as you exit from DDT. Consequently, you should make notes on a copy of your program while using DDT. To exit from DDT at any time, just type ^C, twice if necessary.

Although DDT has closed location PRINT, it has not moved its pointer. The location currently referenced by the pointer is represented by the symbol . (a period). So to see the new contents, instead of typing

```
PRINT/
```

it is only necessary to type

```
./
```

Suppose now you want to examine the contents of the line following the one labeled PRINT in your program. This is done by

```
PRINT+1/
```

similarly, the line before the one labeled PRINT is opened and its contents typed out by

```
PRINT-1/
```

You can examine any line in your program in this way, by counting octally from a label.

Occasionally you may find that when DDT types out the contents of a word they look very different from what appears in your program. For example, if OUTCHR 6 is in your program, DDT will insist on TTCALL 1,6 . If this happens, just use the index of instructions to check that the two are in fact identical. DDT might even insist that you enter new contents in the form it prefers.

If you ask DDT to reference a location that is not there (for example, if there is no location named PRINT), it will reject the request by typing the symbol U ; this indicates that the name you typed is Unidentified. Similarly, DDT will not allow you to enter new contents that have no meaning. If you tried to change the contents of a word to IDIVO N,12 because of a typing error, DDT would again type a U . In some installations the U would be typed after you entered the new contents and a ↵ ; in others, it would appear as soon as you committed yourself to an undefined symbol by typing the space after the letter O.

The effect of the RUBOUT or DELETE key also depends on the installation. In some you may use

it in the familiar way. But in others, DDT will respond to this key by typing XXX. If this happens, you should retype the command from the beginning.

After, should you so wish, modifying a word, you may close it not only with ↵ , but also with LINE FEED (↓), up-arrow (^), TAB (→\|) or backslash (\ ).

When working through a program line by line, ↓ is the most useful word closing instruction: it also moves the pointer on to reference the next location, opens it, and types out its contents.

Before typing out the contents of this next word, DDT will type out its name, followed by / . The name may not be the one you would have chosen from labels in your program. It might be something like DDTEND+25 , or perhaps just a number, like 6234 . DDT calls the first line of your program DDTEND, because it is loaded into core immediately following the end of the DDT program. If a location is given a number, then that is the actual location used in your memory space. You can ask DDT for the contents of any octal numbered location you please; but if you ask for a location that the system has not allocated to you, you will just get a ? in response. The accumulators can be referenced by their numbers, 0 through 17, or by names given to them in your program.

When DDT types out a name defined by your program, it puts the symbol # after it. We have excluded this here, to avoid complicating the notation.

The word closing instruction ^ goes in the direction opposite to ↓ . It sets the pointer to reference the previous word, opens that word, and types out its contents.

The action of the word closing instruction →\| is a little more complicated. Suppose we have typed in

```
LAB1/
```

and that DDT has responded with

```
JRST LAB2
```

as being the contents of location LAB1. Perhaps we are interested in checking through the branch of the program that goes off to location LAB2. If so, we close LAB1 with the →\| instruction. This sets the pointer to location LAB2, opens location LAB2, and types out its contents. If DDT now responds by typing

```
LAB2/        CAME   AC,MEM
```

we can if we wish again use the word closing instruction →\| . Its effect will be to set the pointer to location MEM, to open location MEM, and to type out its contents. The effect of the →\| instruction is to move operations to the last address typed. If MEM contains 0 and is closed with →\| , the pointer is set to accumulator 0, which is opened, and its contents typed out. (But note that the right half word alone determines the new location; indexing and indirect addressing are not taken into account.)

Suppose now that after closing LAB1 with →\| , thereby having the contents of LAB2 typed out, we change the contents of LAB2 in the following way

```
LAB2/        CAME   AC,MEM      CAME   AC,MEM+1 →|
```

closing LAB2 with →\| also. In this case, the last address typed is MEM+1; accordingly, the pointer is set to MEM+1, it is opened, and its contents are typed out. Observe that the last address typed is the one to which the pointer is moved by →\| , whether that address was typed by DDT or the user.

Note that if location PRINT is closed with contents IDIVI N,12 by →\| , the pointer is set to accumulator 12, which is opened and has its contents typed out.

Like →\| , \ opens the last address typed, and types out its contents. However, \ does not move the pointer. Suppose the "location chasing" of the above example were done with \ in place of →\| , opening successively LAB1, LAB2, MEM, and 0. Now type ↵ to close location 0, followed by ./ , and see that the pointer is still set to LAB1.

The character / may also be used as a word closing instruction. It is identical in its effects to \ in

all respects, save one. When a word is closed with / , modifications to the contents typed in by the user are not put into effect.

Later on, when you are using DDT to ascertain how a program performs in action, you will need to check whether a given location contains the data it is supposed to. However, the / instruction always causes DDT to type out the contents of a word in the form of a MACRO-10 instruction, using symbols defined by the user, if it possibly can. If, after such a typeout, you want to see the octal numerical equivalent, just type an = sign. This will always give you the numerical value of whatever symbol was last typed out, whether by DDT or the user. So if you type .= you will immediately get the number of the location to which the pointer is set. This is one case where you would prefer to have typeout in symbolic form. You can have whatever was last typed out (by DDT or the user) repeated in symbolic form by typing the symbol ← (this is _ on some terminals).

Now we are ready to learn how to let DDT execute our program. Commands relating to DDT's carrying out instructions of the program are always of the form $ (ESCAPE) followed by a letter of the alphabet. The command to begin execution is $G. It is, however, not generally very helpful to issue this command without certain preparatory steps, as the program would just run straight through as if it had been EXecuted, DDT would exit, and nothing would be gained.

A simple way to use $G effectively is to check whether a program works from a certain point on. The instruction $G alone starts execution from your program's start address (that is, the location specified in the END statement). However, if you type an address (symbolically, or as an octal number) immediately before $G , execution will start from that address. Suppose, for example, that you want to begin by checking that the printout routine works properly. So after putting test data into any appropriate locations, the command PRINT$G could be used to start operations from the location called PRINT in your program.

Note that if the printout procedure is written as a subroutine whose first location is called PRINT, then the first instruction of the subroutine will be in the following location. So the proper command in this case is PRINT+1$G.

This is still not an efficient way to use DDT because there will be an exit after every trial run. So before starting the program at any location, you should make sure that execution will stop at a convenient point. This is done by setting *breakpoints.* We shall discuss first how to set and remove breakpoints, then how to use them.

A breakpoint is set at a location by typing the address of the location (symbolically or as an octal number), then the two characters $B . DDT will respond by skipping a few spaces. Up to eight breakpoints may be assigned in this way. They are recognized by DDT as 1B through 8B ; when a breakpoint is set, the smallest available number is assigned to it. Suppose that the third breakpoint specified was at PRINT+1; then it may be removed by typing 0$ followed immediately by 3B . Typing $B removes all breakpoints.

The time for setting or removing breakpoints is before starting execution with an $G instruction, or after DDT has completely performed one of the $P or $X instructions discussed below.

One restriction should be observed. Breakpoints should not be set at a location that will be referenced by an instruction using indirect addressing. Thus, if the instruction JRST @LAB is going to be performed, a breakpoint may not be set at location LAB. Essentially, a breakpoint may not interrupt an effective address calculation.

We can begin by setting a single breakpoint at the start address; if this bears the name START we type

```
START$B
```

and DDT will respond by skipping a few spaces. Now we start execution by typing $G . DDT will execute instructions until it reaches a breakpoint; it then stops before executing the instruction at the breakpoint.

In this case, since there is a breakpoint at the start address, DDT will stop before executing any instructions at all, and will tell us what is going on by typing out

```
$1B>>START
```

Now we can work through the program line by line. The command $X instructs DDT to execute the next instruction. With this command, DDT will print out the contents of the locations referenced by the instruction, indicate if the instruction performs a skip or a jump, then print out the line of the program that is next in order of execution. At this stage we can examine and modify the contents of any locations, and set or remove breakpoints. Then another $X will get the next instruction executed, and so on. At any time, instead of typing $X we can type $G to start the program again from the start address. Modifications made to the contents of locations remain in force.

If the instruction being executed is INCHRW, the user must type a single character as input, after giving the $X command. If the instruction is INCHWL, a single character followed by ↵ should be typed. Be careful not to type characters for input when DDT is expecting an instruction, and vice versa.

If you enter *n*$X where *n* is a number, DDT will perform the $X command *n* times over. The number *n* will be interpreted as octal, unless it contains the digits 8 or 9, in which case it is interpreted as decimal (so 9 is one more than 10 in this case).

If the next instruction is a JSR (or JSP, JSA, PUSHJ) to a subroutine that has already been debugged, the instruction $$X is very useful. This tells DDT to go on performing $X commands until the contents of PC attain either one more or two more than their current value. This can be used to execute a subroutine, and stop before the first instruction to be executed on the return. (If a JSA passes more than one parameter, they must be passed by ARG operators to make use of this facility, with the return being JRA AC,(AC) or JRA AC,1(AC) .)

Another way of passing rapidly over a sequence of instructions is to use the $P command. This causes DDT to execute instructions, starting from wherever execution last stopped, until it reaches an instruction at which a breakpoint has been set. Again, DDT will stop before executing the instruction at the breakpoint. Nothing will be typed out until the breakpoint is reached (unless instructions like OUTCHR are executed, or a program error causes an exit and error message).

To make proper use of DDT, the user must be able to modify the contents of locations containing information in various forms: numerical data, ASCII test, program instructions, and so forth. It is also helpful to be able to get such information typed out by DDT in the appropriate form. There are instructions to DDT enabling the user to set *type-in mode*: that is, to specify how DDT is to interpret what the user types in at the terminal. There are also instructions to set *type-out mode*: that is, to specify to DDT the form in which it is to type out information. It is important to realize that type-in mode and type-out mode are wholly independent of each other.

We have already encountered the instructions = and ← (or _) which change type-out mode just until the last typeout is repeated. More generally, type-out mode may be set by typing $ followed by one of certain letters of the alphabet. $S instructs DDT to endeavor to type out everything as a Symbolic instruction. When the $S mode has been selected, $R will cause the address part to be typed out symbolically (as in JRST START); this is the initial setting for DDT. In $S mode, $A causes the address part to be typed out numerically.

Other useful type-out modes are: $C, as numerical Constants; $F, as Floating point numbers; $T, as ASCII Text; $6T, as SIXBIT Text; $H, as Half words, separated by ,, (the right half is interpreted by $R or $A as in symbolic mode).

After typing any of these instructions to change the mode, the instruction ; will cause whatever was last typed (by DDT or the user) to be retyped in the new mode.

In an example that we ran, K denoted accumulator 7, N denoted accumulator 12, and LIST assembled into location 42400 . Some typeouts for a certain line of the program were:

```
$S$R;       HLLZM     K,@LIST(N)
$A;         HLLZM     7,@42400(12)
$F;         -4.6079673E+15
$T;         ROPE
$C;         512372,,42400
```

Initially DDT types out all numbers in octal notation. The radix (base) may be changed to *n* by

the command $nR , where *n* is given as a *decimal* number. After $10R , so that the base is ten, DDT will put a period after every number it types out.

Changing mode or base in the way indicated above effects only a temporary change, which lasts until the user next types a ↵ . A permanent change is effected by typing $$ instead of $, then the appropriate letter code. DDT will skip a few spaces in response. It is still possible after this to change the mode temporarily by $ followed by the code. But now typing a ↵ will change the mode back to the one last set by a $$-type command.

For type-in, numbers not including a period are read by DDT as octal numbers, regardless of the radix setting for typeout. However, a number that could not be read as octal because it contains a digit 8 or 9 will be regarded by DDT as a decimal number.

Numbers followed by a period, with no digits following the period, are read by DDT as fixed point decimal integers. Numbers incorporating a period followed by at least one digit are regarded as floating point decimal numbers; they may be followed by the symbol E and a positive or negative exponent. The exponent must be a number not including a period; it will be read by DDT according to its normal fashion, as described above.

ASCII text may be entered into a word by typing the symbol " , then the desired text between delimiting characters. Thus typing

"/ROPE/ ↵

enters the text ROPE into whichever word was open, and closes the word. This procedure enters the text left justified in the word, as is done by an ASCIZ or ASCII statement. A single character may be entered right justified in a word (as is done by the INCHWL command) by preceding it with " and following it with $. Thus "X$ ↵ enters X in this fashion, and closes the word. (The $T typeout mode is designed to be effective with text entered in either of these ways.)

Half words may be typed in using ,, as separator. We dealt with type-in of symbolic instructions earlier in this section.

The features of DDT that we have so far described are adequate for most purposes, and indeed many programmers never use more than this somewhat limited set of commands. The more advanced procedures we now introduce may be absorbed at greater leisure; they by no means exhaust the powers of DDT.

You will have observed that, although DDT is initialized in fully symbolic type-out mode (SR), it does not use any symbol defined by the user in its type out, until the user has at some stage typed that symbol. When the user types a symbol, DDT searches the symbol table for it, and will thereafter use it. The entire symbol table can be made available to DDT at once by typing the title of the program followed by $: . So if the program contains a TITLE statement giving it the name TEST, the symbol table is opened to DDT by typing

TEST$:

If there is no TITLE statement in the program, the assembler supplied title .MAIN should be used as the program name.

The symbol table may be amended by issuing commands to DDT. Typing a symbol followed by $K instructs DDT to discontinue using the symbol in its typeout. Thus, if AC has been defined, typing

AC$K

will prevent DDT from using the symbol AC in typeout. However, DDT will still recognize AC if the user types it.

Using $$K in place of $K deletes the preceding symbol entirely from the symbol table. Thereafter, DDT will treat it as undefined.

New symbols may be added to the symbol table. A symbol is introduced and given a specified numerical value by typing first the value, then < , then the symbol name, then : . For example, to introduce the symbol AC and give it the value 17, type in

17<AC:

DDT will skip a few spaces in response. A new symbol may be introduced by reference to an old one. Now that AC has been defined and set equal to 17, typing in

```
AC*2+3<X:
```

defines the symbol X with the value 41 (octal). And

```
X-AC*2<M:
```

now defines M with the value 3. (Division with the remainder discarded is represented by ' , since / has another meaning.)

If a symbol has already been defined, the above procedure can still be used, and will serve to give the symbol a new value. For example,

```
M+1<M:
```

will now set M=4 .

A symbol may be introduced and given as value the current location of the pointer, by typing the symbol followed by : . Thus, typing

```
LABEL:
```

will give the word at which the pointer is set the new name LABEL. This is very convenient when putting extra instructions into a program while debugging it. The location following the last instruction in your program is given the name PAT.. by DDT, and locations from PAT.. onward may be used for further program instructions. Suppose the pointer has been set to PAT.. and the user types

```
LABEL:
```

thereby giving PAT.. the new name LABEL. This location may now be opened, and a new program instruction typed in. If LABEL is closed with ↓ , another instruction may immediately be entered at LABEL+1, and so forth. An instruction elsewhere might be amended to read, perhaps, JRST LABEL ; and a new routine will have been created.

If it is desired merely to execute a single instruction that is not already in the program, this can be done at any time by typing the instruction, then commanding its immediate execution with $X ; for example

```
PUSHJ P, LABEL$X
```

There can be problems with the use of $X to carry out a single instruction of the program when that instruction causes monitor intervention; as, for example, when pushdown list overflow traps to a routine supplied by the user. In such a case, it is best to set a breakpoint at the first instruction of the trap servicing routine, and use $P to reach it.

For each of the eight breakpoints, the DDT program maintains a block of three locations. The first location in each block is called $*n*B , where *n* is the breakpoint number as a decimal numeral. The first character in the name of this location, $̈ , is just the dollar sign; in this section only, we are placing the diacritical mark ¨ over it to distinguish it from ESCAPE.

The first location in each such block contains the address at which the breakpoint has been set, in its right half. So if a single breakpoint has been set at START, then

```
$̈1B/
```

will yield typeout of START . The contents of this word are zero if the corresponding breakpoint is not set.

In general, $*n*B is a command to DDT to put the previously typed in quantity into location $̈*n*B . So typing

```
LAB2$4B
```

sets the fourth breakpoint at location LAB2. So also does typing

```
LAB1,,LAB2$4B
```

since only the right half of location $\ddot{\$}$4B determines the address of the fourth breakpoint. But when a breakpoint is reached, DDT types out the contents of the word whose address is in the left half of the first word of the block maintained for that breakpoint. So now, every time the fourth breakpoint is reached, the contents of LAB2 will be typed out by DDT. (A zero left half in the first word of the block specifies no typeout; so the contents of accumulator 0 cannot be automatically typed out in this way.)

The next thing that DDT does on reaching breakpoint *n* is check whether location $\ddot{\$}n$B+1 contains zero; if it does not, DDT executes the contents of that location as an instruction, by performing

```
XCT     $n̈B+1
```

So location $\ddot{\$}n$B+1 can be opened and set up with an instruction (such as a jump to a special routine) to be performed whenever the breakpoint is reached.

Finally, DDT goes to location $\ddot{\$}n$B+2. This contains the *proceed counter.* DDT decrements its contents by 1, and stops operations if it is now negative or zero; this will be the case unless the user has directly amended the contents of $\ddot{\$}n$B+2. Otherwise, DDT continues until it next reaches a breakpoint, where it again goes through this whole procedure. So if, for example, the proceed counter is set to 6, operations will not be stopped at that breakpoint until it is reached for the sixth time.

The proceed counter may be set by opening location $\ddot{\$}n$B+2 and inserting the desired count. Alternatively, if DDT has stopped at breakpoint *n* , the command

```
k$P
```

where *k* is a number, will set the proceed counter at that same breakpoint to *k* , and also issue a $P command.

Effectively, on reaching breakpoint *n* , DDT performs the following sequence of operations with regard to locations $\ddot{\$}n$B+1 and $\ddot{\$}n$B+2 :

```
SKIPE   $n̈B+1
XCT     $n̈B+1
SOSG    $n̈B+2
```

The next instruction in the DDT program jumps to a routine that stops operations and awaits further commands; the next but one instruction in the DDT program jumps to the routine to continue executing instructions of the user's program. Thus, $\ddot{\$}n$B+1 and $\ddot{\$}n$B+2 can be set up to make stopping at the breakpoint conditional on practically any desired set of circumstances.

APPENDIX B

TECO

TECO is a text editor for ASCII text. TECO recognizes two levels of structure within an ASCII file: the *line,* and the *page.*

A line, as far as TECO is concerned, is any string of characters that begins at the beginning of the file or after a character denoting the end of a line, and that ends with the first occurrence of a character denoting the end of a line. We have already encountered the LINE FEED character (^J - ASCII O 12) as denoting the end of a line. Note that the CARRIAGE RETURN character (^M - ASCII O 15) does not signify the end of a line. Two other characters can also be used to signify the end of a line: VERTICAL TAB (^K - ASCII O 13) and FORM FEED (^L - ASCII O 14). The precise physical effect of these depends on the characteristics of the terminal being used. If they have no effect at all, the monitor needs to be informed that the terminal has no form feed capabilities, by the command

SET TTY NO FORM ↵

the monitor will then respond to VERTICAL TAB by transmitting four line feeds, and to FORM FEED by transmitting eight line feeds. Neither of these characters sends the carriage to the beginning of the line; only CARRIAGE RETURN does that.

Observe that a TECO line may be of any length. When a very long line is typed out at the terminal, the carriage will automatically return and move down a line when there is no more room on paper or screen. This is a physical response by the terminal, occurring as soon as the position of the carriage reaches the maximum permitted width. This response does not cause a ↵ to become part of the file, as can be seen by varying the carriage width with the command to the monitor

SET TTY WIDTH *number* ↵

If your file contains the letter A ten thousand times, then ↵ , then B ↵ , it will be typed out at the terminal in many physical lines. Nevertheless, a single L command to TECO will set the pointer between ↵ and B.

TECO's pointer is always regarded as being positioned between two characters, or before the first character in the file, or after the last character in the file. It is *never* regarded as being positioned *on* a character. The pointer is a byte pointer established by the TECO program. It is regarded as being

positioned between the character that it would access by an LDB instruction and the character that it would access by an ILDB instruction.

We are already familiar with some TECO commands that change the position of the pointer. The pointer may be moved to any position within the text currently contained in TECO's editing *buffer*. We shall discuss the buffer below.

Commands to reposition the pointer within the buffer fall into two categories: character oriented and line oriented. We have encountered the C command: *n*C moves the pointer *n* characters forward. (In this section, letters *m* and *n* will always denote positive integers.) Numbers specified to TECO are regarded as decimal; if an octal number is to be entered, it should be preceded by ^O (up-arrow, then O; not CONTROL-O).

The command −*n*C moves the pointer *n* characters backward. This may also be achieved using the R command; *n*R is equivalent to −*n*C , and −*n*R to *n*C .

The command *n*J positions the pointer just after the *n*th character in the buffer. With other commands, if no number is specified, 1 is assumed. With this command, however, J is interpreted as 0J , and moves the pointer to just before the first character in the buffer.

No command to reposition the pointer can send it forward beyond just after the last character in the buffer; nor backward beyond just before the first character in the buffer. Any attempt to move the pointer further than this with a C, R, or J command will result in an error message, and *the pointer will not be moved at all.*

The symbol Z is interpreted by TECO as representing the total number of characters in the buffer. It may be used in place of the numerical argument to a C, R, or J command. For example, the command ZJ moves the pointer to just after the last character in the buffer; so does JZC. After either of these commands, ZR moves the pointer to just before the first character in the buffer.

The symbol . (a period) is interpreted by TECO as representing the number of characters in the buffer preceding the position of the pointer. Thus, .J leaves the position of the pointer unchanged. TECO will perform arithmetical operations on numerical arguments, so the command .+*n*J has the same effect as *n*C. Here the numerical argument for the J command is taken as .+*n* ; that is, the number of characters from the beginning of the buffer, plus *n* . Similarly, .−*n*J is the same as the *n*R command.

The only line oriented command to reposition the pointer is L. This command always moves the pointer to just before the beginning of some line. With argument 0, the pointer is set to the beginning of the current line. The effect with positive and negative arguments is already familiar. If an attempt is made to move the pointer beyond either boundary of the buffer with an L command, the pointer is moved as far as the boundary; there is no error message.

The type out command T is both character and line oriented. We are familiar with the line oriented form, in which *n*T types out from the pointer up to the *n*th following end of line character; −*n*T types out the *n* preceding lines, plus the current line up to the pointer; and 0T types the current line, up to the pointer.

If T is preceded by two numerical arguments separated by a comma, it is regarded as character oriented. The command *m*,*n*T will type out the *m*+1th through the *n*th characters in the buffer; *m* must be smaller than *n*. A nice use for this form of the command is with the . symbol. The command .,.+*n*T will type out the *n* characters immediately following the pointer, while .−*n*,.T will type out the *n* characters immediately preceding the position of the pointer.

The whole buffer can be typed out by the 0,ZT command. However, TECO recognizes the symbol H as representing 0,Z ; so HT gets the whole buffer typed out.

The T command never moves the pointer. After the command string

```
J10L0,6T$$
```

the pointer is still set to the beginning of the eleventh line, although the T command has just had the first six characters in the buffer typed out.

The K command almost precisely parallels the T command. Whatever a T command, with one or with two numerical arguments, would type out, it will be deleted by the same command with K replacing T. The difference between K and T lies in the effect of K on the pointer. The command *n*K

does not affect the pointer at all; but $-n$K , 0K and m,nK all move the pointer to just after the last character preceding the deleted text.

The D command can also be used to delete characters. The commands nD and $-n$D delete the n characters following and preceding the pointer. nD does not affect the pointer, but $-n$D moves the pointer to just after the last character preceding the deleted text.

The I command for insertion of text needs little further comment. The pointer is moved to just after the last character inserted. The →| command (this command is just the TAB character) differs from I only in that the →| itself becomes part of the text.

The I and →| commands cannot be used to insert the $ (ESCAPE) character or most of the CONTROL characters as part of a text. (However, they can be used to insert the familiar CONTROL characters whose function is to move the carriage horizontally or vertically.) Any character may be entered into a text by typing the ASCII code for the character, followed by I, then $ to terminate the command. Only one character at a time may be inserted in this way. The pointer is positioned just after the inserted character. Thus, 27I$ will insert $ as a text character. Note that decimal notation is used for the ASCII code; alternatively, ^O33I$ may be entered.

The \ command is preceded by a single numerical argument. The effect of this command is to insert the numerical argument as text. Like all TECO commands, the \ command is not performed until $$ is entered. Actually there is no difference between, say, I123$ and 123\ ; each inserts 123 as text. However, \ will insert the decimal representation of any numerical value currently presented to TECO. Thus, IZ$ will insert the character Z into the text; but Z\ will insert into the text the decimal representation of the number of characters in the buffer. For example, suppose that the buffer contains

THE#QUICK ↵

where the symbol # denotes a space. Then the command string

6CZ\$$

will leave the buffer containing

THE#QU11ICK ↵

and the pointer will be positioned just before the letter I . If the command C.\$$ is now issued, the buffer will contain

THE#QU1119CK ↵

The S command searches for the given character string within the buffer. An unsuccessful search results in an error message, and the pointer is moved to the beginning of the buffer. With a positive numerical argument n, nS will search for the nth following occurrence of the given character string. S alone is the same as 1S.

The command FS to search for a character string within the buffer and replace it with another is familiar.

The commands :nS and :FS do everything that is done by the corresponding commands without the colon, except that no error message is printed if the search fails. In addition, these commands make available a numerical argument, which may be used. The numerical argument is −1 if the search is successful, 0 if it fails. For example, suppose THRU occurs in the buffer just nine times. Then the command string

:8STHRU$\$$

will insert −1 into the text after the eighth occurrence of THRU, because this is the position of the pointer after the search is successfully completed. But

:10STHRU$\$$

will insert 0 into the text at the beginning of the buffer, because this is the position of the pointer after the search fails.

The :*n*S and :FS commands are said to *return a value.* Note that if the returned value is to be used, it must be used forthwith. The command string

```
:10STHRU$$
```

loses the returned value when the $$ is typed to have the search command performed. A \ command will not now have the desired effect. Also, an attempt to insert the returned value at the beginning of the buffer by

```
:8STHRU$J\$$
```

will fail, as the J command uses the value returned by the :8S command as its own argument; the effect is −1J , resulting in an error message.

The command \ does not destroy the existence of a returned value, but may not leave it unchanged. This might have an unexpected effect on the next command. So after \ , a command that takes a numerical value should be preceded by $; alternatively, the argument can be made explicit, as with 1T , or 0J .

The concept of a returned value is crucially important in TECO. Note that Z and . may be regarded as commands whose sole function is to return a value.

A returned value may be checked with the = command. This instructs TECO to type out the decimal value of the argument to the command. Thus, after 10=$$, TECO will respond with 10 ; after Z=$$, TECO will respond with the number of characters in the buffer; and after :STHRU$=$$ TECO will respond with −1 if the search was successful, 0 if it failed.

All the TECO commands so far considered edit only the text in TECO's buffer. When TECO is used to access an already existing file, the first page of the file is read into TECO's buffer. Encountering a FORM FEED character (denoted here by ▽) signifies the end of a page to TECO. The ▽ character itself is not considered part of the page, and is not read into the buffer.

If no ▽ character is encountered, text is read in until either the whole file has been read, or the buffer has no further capacity. The buffer is initially large enough to hold the contents, double line spaced, of a regular size paper page, or of a display screen. The A (append) command expands the buffer, and reads into it a further page, if now there is room. So a sufficient number of A commands will get the whole file read into the buffer at once. However, this is often not a good idea, as any page structure in the file is lost in the process. When the A command reads in a new page, it discards from the file the ▽ character separating the pages, thus combining pages into one large page. The page structure can be very useful, especially in long files of text. It facilitates the formatting of text: the line printer starts a new page on receipt of a ▽ . The RUNOFF text formatting program of the DECsystem-10 uses ▽ as a separator of physical pages, so A commands damage RUNOFF output. In addition, it is easier to find one's way around in a long file if it is page structured, using page oriented TECO commands.

To understand how TECO handles pages, we must introduce the concepts of *input file* and *output file.* When TECO is called for an existing file, it opens the current version of the file for input. For output it opens a new file. We are familiar with the input command A , which, if repeated often enough, will input the whole file into TECO's input/output buffer (and destroy the page structure). The only output command that we have encountered is EX , which in fact performs both input and output functions. This command transfers the contents of the buffer, and any succeeding pages of the input file, to the output file. (The page structure of any text following the buffer contents is unaffected.)

The EX command also closes the input and output files, gives the output file the name previously borne by the input file, and gives the name of the input file the new extension .BAK . Since all this is done by a single instruction, it is easy to be unaware that separate input and output files are involved.

When TECO is invoked, as with the MAKE command to the monitor, to create a new file, an output file with the specified name is opened; there is in this case no input file.

To create a file with page structure, just type a ▽ wherever a new page is desired, as part of the text string to be inserted with an I command. A file of many pages may be created in this way;

TECO will expand its buffer to hold them all. Alternatively, each completed page may be sent to the output file before going on to type in the contents of the next page. The TECO command P sends the current contents of the buffer to the output file, and clears the buffer. When there is also an input file, P reads the next page of it into the buffer.

Suppose the following command string is typed

IABCD↵▽$PIEFGH↵▽$$

Of course when ↵ and ▽ are entered, the corresponding effect will register at the terminal; for simplicity of notation we have not reproduced this here.

The command HT will now get EFGH↵▽ typed out, because the P command sent ABCD↵▽ to the output file and emptied the buffer. None of the TECO editing commands will recover pages already sent to the output file.

Let us now suppose that the file has been closed with the EX command, and has been reentered for editing with the monitor command TECO. If we now issue the command HT to type out the entire contents of the buffer, the result will be ABCD↵ . Note that the ▽ is not in the buffer. A flag is set to indicate that a ▽ has been found; the command P checks this flag, and if issued now will send ABCD↵▽ to the output file. Since P also reads in the next page of the input file, the command HT now yields type out of EFGH↵. The only direct way to have your whole file typed out, with its page structure intact, is with a monitor command: TYPE, with the file reproduced at the terminal; or PRINT, with the file reproduced by the line printer. Otherwise, one can use the TECO command ^L (up-arrow, then L), which instructs TECO to issue a ▽ . Thus

10<HT^LP>EX$$

will type out a file of ten pages, putting in the ▽ after each page, and exit from TECO.

The P command may have a positive numerical argument: *n*P sends *n* pages of the input file (starting with the page currently in the buffer) to the output file, then reads a further page into the buffer.

The P command effects both input and output. Output alone is accomplished by the PW command, which may have a numerical argument. This command sends the contents of the buffer to the output file, and appends a ▽ regardless of whether one was originally there or not. The contents of the buffer are left unchanged. So if a file of ten pages has been entered for editing, the command 10P will send the whole input file to the output file; but 10PW will send ten copies of the first page of the input file to the output file.

Input alone is accomplished by the A command, or by the Y (yank) command. Neither of these may have a numerical argument; but *n* repetitions may be commanded by *n*<A> or *n*<Y> . The Y command empties the buffer, then reads in the next page of the input file. It is important to be aware that Y *does no output*; the page currently in the buffer is merely discarded. Thus, PWY is equivalent to P, except that the former command will append a ▽ to the buffer contents even if none was there before. The monitor command TECO automatically causes a Y to read in the first page of the file.

There are search commands that do not discontinue their efforts at the end of the buffer. If N is used in place of S , or FN in place of FS , successive P commands are automatically executed until the text is found. Note that if an N or FN search fails, the whole file has been output; an error message will appear, and if further editing is needed the file must be closed with EX and reentered with TECO. The N command may have a positive numerical argument, as with S ; both N and FN may be modified by a preceding colon, as with S and FS.

The N command causes both input and output—the latter by implicitly generating P commands. The ← command (this is _ on some terminals) is similar to N in all respects, save one: it performs no output. Where N generates a P command, ← generates a Y command. Thus, ← may be used for discarding all pages of a file before the given character string is found.

If a character string is split across two pages, no search command will detect it.

Any TECO command string may be repeated any number of times by placing it within angle brackets <...> . A positive numerical argument may be given to specify the number of times the command string is to be carried out. This argument may be a value returned by the previous TECO command. If there is no argument, the command string will be iterated indefinitely. For example, suppose that the buffer contains THE ↵ . Then the command

```
<ST$TSX$>$$
```

will cause the text HE ↵ to be typed out indefinitely; the unsuccessful search for X always sets the pointer back to the beginning of the buffer.

Searches within a <...> iteration never yield an error message. According to the official TECO manual, all searches within an iteration are equivalent to searches with the colon modifier; but in fact subtle differences exist. Rather than go into detail, we recommend:

(a) use a colon modified search when a returned value is explicitly required for the next TECO editing command within the iteration;
(b) after any search within an iteration, when a returned value is definitely not wanted by the next TECO command within the iteration, give that command an explicit numerical argument (normally 0 or 1).

The command ; can be used to discontinue an iteration when a search has failed. If X is not to be found in the buffer, then

```
10<SX$1T;>$$
```

will type out the first line in the buffer. The search fails, setting the pointer to the beginning of the buffer, and one line is typed out. Note that the failed search within the iteration could return a value for T , so we command 1T explicitly. Now TECO encounters the semicolon command, and discontinues the iteration because a search has failed within it. The command string

```
10<SX$;1T>$$
```

would yield no type out at all, as TECO responds to the semicolon command before encountering T .

Iterations may be *nested*; that is, contained within other iterations. The effect of the semicolon command is then to leave the iteration in which it is found when a search has failed within that iteration. Control then passes to the next higher level of iteration. For example

```
10<<FSX$Y$;>P>$$
```

will change all occurrences of X to Y in the first ten pages of the file (assuming that initially the pointer was set to the beginning of the file). It will output these pages, and input the eleventh page.

The semicolon command is in fact a good deal more powerful than we have so far indicated. It takes a numerical argument, and will discontinue an iteration if the argument is positive or zero. (This command is used only within an iteration.) The numerical argument for the semicolon command may be returned by any preceding command. What we saw above was the semicolon command responding to the value returned by an S command. The colon modifier is not needed for passing a value to the semicolon command.

The Z command returns a positive value when the buffer is not empty; so −Z returns a negative value in this case. Thus, the command string −Z; within an iteration will cause that iteration to cease as soon as an empty page is encountered. For example,

```
<<FSX$Y$;>P−Z;>$$
```

will successively change all occurrences of X to Y on a page, output the page, and input the next, until it encounters a blank page (or until the end of the file is reached).

It is not possible to use an FS or FN command to replace a character string with a null string

(that is, to delete it) within an iteration. Because $ is the string delimiter, the sequence $$ would have to occur to delimit the null string within the iteration. But $$ instructs TECO to execute the preceding command string, which in this case would result in an error message. For example, if the command string

```
10<FSABC$$>$$
```

were attempted, TECO would try to execute the command string 10<FSABC$$, and would conclude that a < had been typed without a matching > .

The @ modifier should be used in this case. It may be used with any of the commands S , FS , N , FN , ← , and I . If there is a numerical argument, @ is placed before it. And @ may be used with the colon modifier, in either order. The effect of @ is to allow the user to specify the character string delimiter, in the same manner as in a MACRO-10 ASCIZ or ASCII statement. So in our example above, a correct form of the command, choosing / as delimiter, would be

```
10<@FS/ABC//>$$
```

The semicolon command has introduced the idea of making performance of one TECO command dependent on the outcome of another. There is a much more general way of doing this. Any command string introduced by the characters "E and terminated by the character ' (not ") will be performed only if it is preceded by a numerical argument that is Equal to zero. Otherwise TECO will skip over the command string, and resume with the commands that follow it. Normally the numerical argument is the returned value yielded by the previous TECO command.

For example, we can go through a file of a hundred pages writing BLANK PAGE ↵ at the top of every blank page, with the command string

```
100<Z"EIBLANK PAGE ↵$'P>$$
```

The command IBLANK PAGE↵$ is performed only if the value returned by Z is zero.

Later, we can edit this file, and discard all pages on which the legend BLANK PAGE↵ appears, stopping when a genuinely blank page is encountered. The command string to achieve this is

```
<:SBLANK PAGE ↵$"EPW'Y-Z;>$$
```

The search with colon modifier (necessary here) returns the value 0 if BLANK PAGE ↵ does not appear on the page; in this case, the command PW is performed, and the page is output. Otherwise the page is lost when a new page is yanked in with the Y command.

Note that neither this command, nor any of the conditional commands discussed below, should be used to exit from an iteration. Only the semicolon command should be used for this purpose.

The "E ... ' command is one of a large range of conditional commands. They all have the "*x* ... ' format, with *x* replaced by one of a variety of code letters. In each case, performance of the intervening command string depends upon the preceding numerical argument. Examples are

n"N ... '	perform if *n* is Not equal to zero
n"G ... '	perform if *n* is Greater than zero
n"L ... '	perform if *n* is Less than zero

These commands may be nested, with each opening "*x* matched with its closing ' , rather like parentheses in arithmetical expressions.

For example, let us amend our previous command string so that only pages on which BLANK PAGE↵ and nothing else appears, are deleted:

```
!LAB!:SBLANK PAGE ↵$"NZ-."E12R."EYOLAB$''PZ"GOLAB$'$$
```

There are several new ideas in this command string. Look first at the colon search command after !LAB! . Suppose that this search is successful; it then returns a nonzero value (in fact, −1), and so the commands following "N are carried out. We could have used L instead of N. Observe that the matching ' is the last one immediately preceding the P command later in the string.

If BLANK PAGE ↵ is found, the first commands to be carried out are

Z−.

TECO will perform such arithmetical calculations; but observe that calculations are performed from left to right unless parentheses specify otherwise. Thus to TECO, 3+4*5=35 , while 3+(4*5)=23. In the present case, TECO returns the number of characters beyond the pointer. Note that the pointer is now set just after the text BLANK PAGE ↵ . If there are no characters after this text, the next conditional command string is performed.

The first commands in the next conditional string are

12R.

The effect is to set the pointer to just before the twelve characters long text BLANK PAGE ↵ , and to return the number of characters now preceding the pointer. If the returned value is zero, we have ensured that BLANK PAGE ↵ is the entire text on the page. Under these circumstances, and these only, the command string YOLAB$ is carried out. Note how each "E and "N is matched by its closing ' .

The TECO command O is a jump command. The location in the command string to which TECO will jump is given by the label following O and terminated by $. At the jump destination the label must appear, delimited by ! characters.

So the string YOLAB$ discards the unwanted page, and goes to the first command after the label LAB to perform the colon search on the new page.

If any of the conditions in the "E or "N commands are not met, the command string PZ"GOLAB$' is carried out. The current page is wanted, so it is output. Then the buffer count for the new page is checked. If it is Greater than zero, the command OLAB$ is performed; otherwise the whole command string terminates.

A further use for the conditional commands involves the special command *n*A . In this command, *n* can be any number, making no difference to the command; the sole function of the argument is to distinguish this command from the A (append) command. The function of *n*A is to return a value equal to the ASCII code of the next character after the pointer; if the pointer is at the end of the buffer, 0 is returned. Thus, a command sequence that goes through the buffer character by character could be interposed within

!LAB!1A"N ..... OLAB$'$$

to ensure that operations automatically stop when the end of the buffer is reached.

The *n*A command is especially useful for returning a value to be used by the following conditional commands, which relate to the ASCII code represented by the argument:

n"D ... '	perform if *n* represents a digit (numeral)
n"A ... '	perform if *n* represents a letter of the alphabet
n"V ... '	perform if *n* represents a lowercase letter of the alphabet
n"T ... '	perform if *n* represents an uppercase letter of the alphabet

Thus, the command string

J!LAB!1A"N1A"D.,.+1T'COLAB$'$$

will type out all the numerals in the buffer.

It would be tedious, and the source of many errors, if long and complex TECO commands had to be entered at the terminal each time they were needed. Fortunately, TECO has provision for storing character strings (which may be command strings) during an editing session.

The locations made available for storage by TECO are called *Q-registers*. Q-registers are quite independent of the editing buffer; their contents are not affected by any of the commands so far introduced.

There are thirty-six Q-registers. They are referenced by their names, which are the single letters of the alphabet A,...,Z and the ten numerals 0,...,9 . We shall use the dummy letter *i* to indicate that the name of a Q-register is to be specified.

The most elementary use for the Q-registers is to move blocks of text in a file. Suppose that a file is being created in which a particular block of text will occur more than once. The first step is to type the text, thereby entering it into the buffer, within an I command. Now the command X is used to copy the contents of the buffer into a Q-register (any previous contents of the Q-register are lost). X is preceded by one numerical argument, or by two numerical arguments separated by a comma. These specify the area of text, in precisely the same way as with the T command. Indeed, X is very similar to T : X writes text into a Q-register, while T writes text at the terminal. The name of the Q-register must follow X . Thus, 0X*i* copies the text from the beginning of the current line up to the pointer into Q-register *i* . In this case, we would want to store all the text that is in the buffer; the command HXA would store the entire contents of the buffer in Q-register A . Since X affects neither the text in the buffer nor the pointer, the command HK would be needed if the buffer were now to be cleared.

The G (get) command can be used whenever the text is required. G is followed by the name of the Q-register holding the desired text; it inserts the text held in the Q-register into the buffer, starting at the pointer. The pointer is moved to the end of the inserted text, just as with the I command. The contents of the Q-register are unaffected.

The X command could be used to insert command strings into a Q-register. However, this would be awkward, as a \ or @I command would be needed every time an $ had to be inserted. A better way is to use the * command, followed by the name of the Q-register. This command enters the previously typed complete command string into the specified Q-register, destroying any former contents. Thus, if we wanted to store the command string :SXYZ$"EY' in Q-register B , we would first issue this command string to TECO

```
:SXYZ$"EY'$$
```

and then, when TECO typed a * to indicate readiness for further commands, we would type

```
*B$$
```

To store a command string without first executing it, abort the string by following it with ^G^G (CONTROL characters) instead of $$. Then when TECO issues a * , type *B$$, just as before, to store the command string in Q-register B .

The M command, followed by the name of the Q-register, will execute the contents of that register as a command string. The whole of any given command must be in the Q-register if the M command is to work. So I, S, and so on, cannot be split from the text being referenced; and a < must have its matching > in the Q-register. However, a numerical argument may be split from the command that follows it. For example, after the command string

```
IH$.-1,.XAIT$.-1,.XB-2D$$
```

H is stored in Q-register A , T is stored in Q-register B , and the buffer is unchanged. Now the command string

```
MAMB$$
```

will type out the whole of the buffer at any time.

The Q-registers may be used for storing single integers (which must not be too large to fit into a computer word). The command *n*U*i* stores the decimal integer *n* in Q-register *i* , destroying its original contents.

The command Q*i* returns a value equal to that of the integer in Q-register *i*. The command %*i* adds 1 to the contents of Q-register *i* , then returns the new value of the contents.

Thus, we can insert page numbers at the top left corner of every page in a file, stopping when a blank page is encountered, with the command sequence:

```
0UB<-Z;IPAGE $%B\I↵ $P>$$
```

in which we use Q-register B .

User errors are just as likely to occur in TECO command strings as in MACRO-10 programs. TECO offers two debugging aids which let the user trace progress through a command string.

```
0U1
<-Z;
IPAGE $
%1\
I
$
^AGETTING NEXT PAGE
^A
P>
EX
```

FIGURE B.1 A TECO macro to paginate a file.

The ^A command instructs TECO, when it reaches it in the command string, to type out the statement that follows; the end of the statement is marked by a further ^A . The first ^A may be either up-arrow, then A , or CONTROL-A ; but the second one, to terminate the statement, *must* be CONTROL-A . It minimizes effort of memory to use CONTROL-A for both of them. We have illustrated the use of this command in Figure B.1, with a command sequence similar to the one above to paginate a file. This figure also illustrates formatting a TECO command string for ease of reading; ↵ may be used freely for this purpose, since it is meaningless to TECO except within a text string.

The ? command instructs TECO to type out each following command as it is executed. A second ? command instructs TECO to turn off this feature. Note that this second, disabling ? is itself typed out by TECO as it is executed. Thus, TECO responds to ?J?HT$$ by typing J? , then the contents of the buffer.

The monitor commands MAKE and TECO do not always provide a suitable context for performing complex editing tasks using the more sophisticated TECO commands. We conclude with a brief description of the general TECO environment, in which greater control over input and output files is obtained.

The TECO program, like all system programs, is called into core by the monitor R command

R TECO ↵

without reference to any particular file for editing. This gives TECO its minimum requirement of 5K (O 5000 words) of core. If it is anticipated that a file with large pages will be edited, TECO should be called with more core:

R TECO 6 ↵

supplies 6K, giving extra to the editing buffer. The 6 may be replaced with higher decimal numbers, as required.

At any time during a TECO editing session, the user may have one file open for input and another open for output. However, in neither case need it be the same file throughout the session.

The basic command to open a file for input is ER. The name of the file to be opened, with its extension if any, follows and is terminated by $. Thus, to open TEST.MAC for input

ERTEST.MAC$$

If this command is issued when a file is already open for input, that file is closed and the new one opened. A file opened with this command is used for input only, and is in no way altered by TECO. The ER command does nothing but open the file; normally the next command will be Y , to read in the first page of the file:

ERTEST.MAC$Y$$

The command to open a file for output is EW , followed, as above, by the file name, terminated by $. This command creates a new file, and any existing file with the same name will be superseded. If this command is issued when a file is already open for output, that file will be closed (and will contain all text previously sent to it by TECO output commands), and the new one opened.

Suppose for example that we have files FILA and FILB, and that we want to create the new file FILC. This is to consist of the first page of FILA, followed by all of FILB, then the rest of FILA.

First we create FILC and open it for output

```
EWFILC$
```

then we open FILA for input, and yank in its first page

```
ERFILA$Y
```

then output that page to FILC, open FILB for input, and yank in its first page

```
PERFILB$Y
```

Now we output all of FILB to FILC. We use the command ^N (typed either way), which returns the value −1 if the last input reached the end of the file, 0 otherwise.

```
!LAB!P^N"EOLAB$'
```

Once again we must open FILA for input. This time we are not interested in the first page

```
ERFILA$2<Y>
```

All that is now needed is

```
EX$$
```

which we have followed by $$ to cause execution of the entire command string. The EX command automatically writes the rest of the input file in the output file before exiting to the monitor.

The file renaming functions of the EX command occur only when an existing file has been opened for updating with the EB command. The effect of EB is already quite familiar to us. In fact, the sequence

```
R TECO ↵
```

followed by

```
EBFILE.EXT$
```

and finally

```
Y$$
```

is precisely equivalent to

```
TECO FILE.EXT ↵
```

The EB command closes any files previously opened by ER or EW commands.

Once an updating job has been started, further EW or EB commands cannot be issued. However, ER can be used to open a new file for input; text from this file can then be output to TECO's temporary file. Recall that when an EX command is issued, this temporary file becomes the new version of the file being updated.

Complex TECO command sequences are known as TECO *macros*, and are often kept in their own files. Such files are normally given the extension .TEC . The easiest way to write such a file is:

(a) create a file with the desired name, with the MAKE command;
(b) type the desired contents of the file as a command string, but conclude with ^G^G instead of $$;
(c) use the * command to save the command string in one of the Q-registers;
(d) write the contents of the Q-register in the buffer, with a G command;
(e) repeat if further commands are to be included, otherwise exit.

Suppose that the command sequence of Figure B.1 comprises the file PAGE.TEC, and that we

want to use it on the file TEXT. After

```
R TECO ↵
```

without yet opening an output file, we open the command file for input, and read its contents into the buffer

```
ERPAGE. TEC$Y
```

Then we save the command macro in a Q-register

```
HX1
```

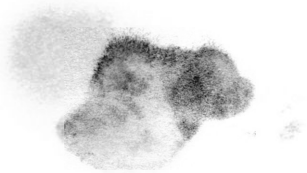
Now we are ready to open the file for editing

```
EBTEXT$Y
```

Note that we must clear the TECO macro from the editing buffer otherwise it will end up as text in the output file; we may as well yank in a page of the file TEXT for this purpose, since it will have to be done anyway. To paginate the file, all that is now needed is

```
M1$$
```

This command returns us to the monitor, since the TECO macro terminated with EX .



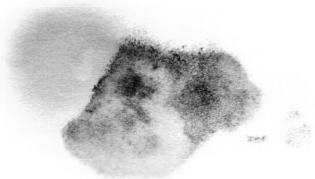
ASCII CODES

CHARACTER	CODE	CHARACTER	CODE
NULL	0	@	100
CONTROL-A	1	A	101
CONTROL-B	2	B	102
CONTROL-C	3	C	103
CONTROL-D	4	D	104
CONTROL-E	5	E	105
CONTROL-F	6	F	106
CONTROL-G	7	G	107
BACKSPACE	10	H	110
TAB	11	I	111
LINE FEED	12	J	112
VERT TAB	13	K	113
FORM FEED	14	L	114
CARR RETN	15	M	115
CONTROL-N	16	N	116
CONTROL-O	17	O	117
CONTROL-P	20	P	120
CONTROL-Q	21	Q	121
CONTROL-R	22	R	122
CONTROL-S	23	S	123
CONTROL-T	24	T	124
CONTROL-U	25	U	125
CONTROL-V	26	V	126
CONTROL-W	27	W	127
CONTROL-X	30	X	130
CONTROL-Y	31	Y	131
CONTROL-Z	32	Z	132
ESCAPE	33	[	133
CONTROL-\	34	\	134
CONTROL-]	35	]	135
CONTROL-^	36	^	136
CONTROL-_	37	_	137

CHAR	CODE	CHAR	CODE
SPACE	40	`	140 *
!	41	a	141
"	42	b	142
#	43	c	143
$	44	d	144
%	45	e	145
&	46	f	146
'	47	g	147
(	50	h	150
)	51	i	151
*	52	j	152
+	53	k	153
,	54	l	154
-	55	m	155
.	56	n	156
/	57	o	157
0	60	p	160
1	61	q	161
2	62	r	162
3	63	s	163
4	64	t	164
5	65	u	165
6	66	v	166
7	67	w	167
8	70	x	170
9	71	y	171
:	72	z	172
;	73	{	173
<	74	|	174
=	75	}	175 #
>	76	~	176 #
?	77	RUBOUT	177

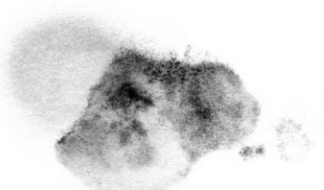
* this code is rarely used

these codes are treated as ESCAPE unless the Monitor command SET TTY NO ALTMODE has been issued

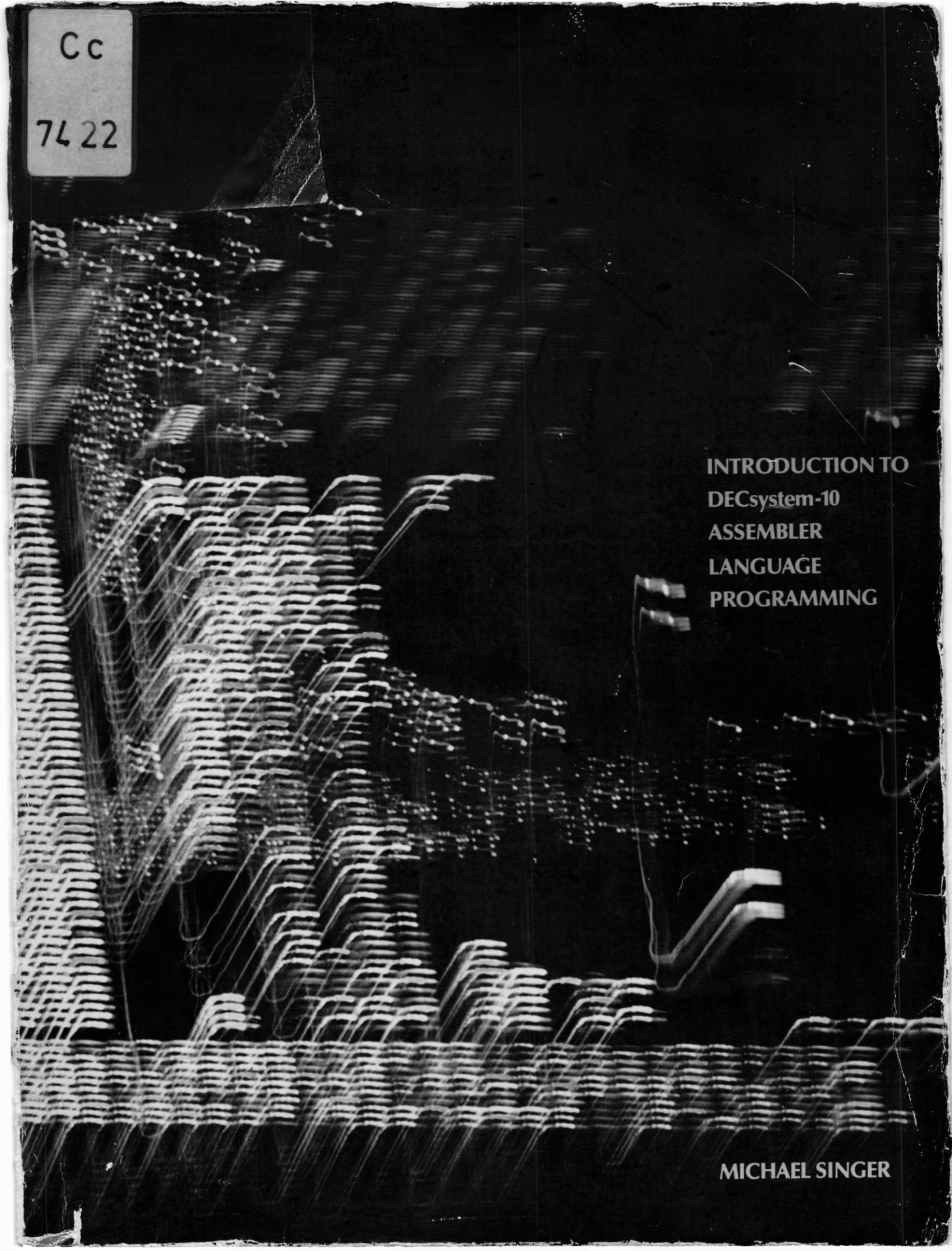


INDEX OF MACRO-10 INSTRUCTIONS

*denotes monitor calls
#denotes pseudo-ops



SUBJECT INDEX



INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

MICHAEL SINGER