INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

MICHAEL SINGER

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

INTRODUCTION TO DECsystem-10 ASSEMBLER LANGUAGE PROGRAMMING

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

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

$\underline{.}$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

$\underline{*}$

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."