PRENTICE COMPUTER CENTRE
University of Queensland

TECO Editing

—A Tutorial Course for PDP-10 and PDP-11 Users

S. H. Algie

MNT—16
APRIL 1978

PRENTICE COMPUTER CENTRE

University of Queensland

TECO Editing

– A Tutorial Course

for

PDP-1Ø and PDP-11 Users

S.H. Algie

1st Edition
April 1978

INTRODUCTION

This book has been written to provide a complete description of the procedures used in editing files with the TECO (Text Editor and Corrector) program. The approach used here differs from that used in the "DECsystem-1Ø TECO Programmer's Reference Manual" (DEC-1Ø-UTPRA-A-D) which describes what is referred to here as TECO-1Ø and in the "PDP-11 TECO User's Guide" (DEC-11-UTECA-A-DN1) which describes TECO-11. These are referred to here as "Reference Manuals" and they are not very effective as teaching texts. This book presents a tutorial course in TECO editing; commands are introduced in the order in which they are needed for effective learning of the TECO language. Practical exercises are given after each of the first six chapters.

TECO is a programming language. The use of TECO for programmed editing is treated in some detail and care has been taken to present it in a way which is familiar to users of high level languages such as FORTRAN, ALGOL and PASCAL.

It is suggested that you should work through the exercises based on the first four chapters *before* you attempt to edit important files. You *can* learn TECO as you go but it will still take a few hours at the terminal to master it and in the meantime you can do a lot of damage. TECO is so powerful that it is a bit like a loaded gun. Be careful! However, once you have mastered TECO you will find that you can perform almost any editing task so quickly and easily that you will wonder how you managed without it.

CONVENTIONS

Throughout this book the symbol (\$) stands for the ASCII character ESCAPE (or ALTMODE); characters transmitted by holding down the CONTROL key while striking another (generally x) are represented by a symbol (↑x) . Throughout the text ASCII characters such as LINE FEED are written in upper case but when they are given in commands and confusion may arise they are written thus: <line feed>. Terminal output is underlined to distinguish it from input typed by the user.

DECLARATIONS

The terms "TECO", "DECsystem-1Ø", "PDP-11", "DECTAPE", "RT-11", "RSTS/E" and "RSX-11" are trademarks of the Digital Equipment Corporation.

The author assumes no responsibility for any loss of, or damage to, computer files which may result from the application of procedures described in this book.

Please inform the author of any errors.

I acknowledge with pleasure and gratitude the willing assistance of my friends and colleagues in the writing and production of this book. I especially thank my wife for her patience.

Steve Algie

CONTENTS

PART ONE

ESSENTIAL TECO EDITING

1. Deleting and Inserting Text.
2. Additional Text Modification Commands.
3. TECO Input and Output.
4. Additional Input and Output Commands.

The four chapters in this part contain sufficient information to enable the user to tackle effectively most interactive editing tasks. Some of the information given here is not essential to the immediate requirement of basic editing but is required for the understanding of the more advanced techniques described in subsequent parts.

1. DELETING AND INSERTING TEXT

Information which is to be input to a computer is usually stored as a file. An exception would be the information which you enter through a terminal keyboard, although if you copy this from a written record this written version might be considered a file of sorts. Decks of punched cards and rolls of punched paper tape are two types of files; the predominant type of file at present is that stored on some kind of magnetic disk. The information stored in such a file may comprise computer programs or data (which is information to be processed by the computer). Data may consist of numeric information or of text; processing includes simply the output of data in a form you can understand. For example, a book could be stored in a file as text data and the computer, operating under an appropriate program of instructions could have this typed out at your command. A sad thing about files is that the information contained in them rarely remains correct, or appropriate, for long. The book mentioned above will probably go into a new revised edition!

What do you do when you want to change a part of a file? Retype the whole file and re-enter it? This approach has little appeal if you want to change, say, just one "X" to a "Y" in the middle of a long program. It might be appropriate if you want to change 95% of the data in a data file.

If you learnt computing using decks of cards you will remember that you could change that "X" to a "Y" by

(1) searching for the line on which it occurs

(2) searching for the card which corresponds to that line

(3) replacing it with an altered card.

You will be relieved to know that editing is possible with disk files too, using TECO. Moreover, the computer does the searching for you.

WHAT IS TECO?

TECO, the Text Editor and COrrector, is a computer program. If you really wanted to, you could write your own (e.g. FORTRAN) program to change every letter X in a file to a Y. If you arranged your program to write the message "what letter do you want altered and what is it to be changed to?" so that when you typed in X,Y the change was made then you'd have written a simple editing program. TECO is just a better one.

TECO is a very powerful program which enables you to accomplish almost any task of editing ASCII files. With TECO you can insert new sections into files, delete sections, rearrange the contents of a file without deleting from or adding to it, merge two files and split a file into sub-files. Moreover, this does not exhaust the powers of TECO. Its features allow the user to make the most complex conditional alterations.

For example, you could write a report, or even a book, and using the computer store it as a file. You could then, using TECO, change a paragraph here or there, rearrange the sections, add sentences, change words and then with one (complex) command have the revised version typed out in pages of, say 60 lines with each line containing as close to, say, 70 characters as possible but without splitting words across two lines. Unfortunately TECO does not correct spelling errors automatically.

TECO is so useful that it is made part of the set of system programs in many computers. If it is present it can be run by giving the *system command:*

. R TECO (DECsystem1Ø and RT-11) *or*
RUN $TECO (RSTS/E) *or*
TECO (RSX-11)

This loads and runs the program but instead of asking a question like that written above, TECO indicates that it is ready to receive instructions (or commands) from the user by typing an asterisk:

*

The many commands available in TECO allow you to instruct it to read a file, alter it (e.g. change every X to Y), and write the correct file. If TECO understands your command it will execute it; if not it won't and will print an error message instead. TECO will, of course, execute commands as you actually give them, which is not always the same as how you intended to give them.

Thus, there is a price to pay for TECO's convenience. You can drop a deck of cards and it's a nuisance. With TECO you can cause yourself real grief, especially if you let your attention slip or panic. Imagine the situation: you've unintentionally deleted your file, you aren't sure what you should do next and there's someone standing behind you and waiting impatiently (and unsympathetically) for you to finish. A disaster is about to occur. This won't happen to you if you study these notes carefully. You probably won't find the advice that you should be very careful any more helpful! Unfortunately there is no more practical advice that can be given.

One reason for this is TECO's very strength. The commands which you will give to change the contents of your files are generally in the form of numbers and letters, the letters usually being one or two letter abbreviations of the name of the command (e.g. D for delete). However *almost every key* on the terminal acts as a command. If you don't get in the habit of checking to see what you actually typed, as distinct from what you thought you typed, you will change your file in unintended ways. "Murphy's Law" applies in this case: the unintended command you give will be the one which is hardest to correct!

You do need to think while editing. If you change every X to a Y you must realize that you can't always undo this by subsequently changing every Y to an X! This is nothing to do with TECO; the same would happen with the FORTRAN example given above.

Another problem is that, like FORTRAN, TECO has been developed over a number of years and like FORTRAN retains a number of features which are redundant. It also has features which are better avoided. Some of these will be mentioned in these notes but the preferred forms are emphasized.

A final word of encouragement: The concepts involved in TECO are of general application in many editing programs. When you've learned TECO you'll have little trouble in learning another if it should be necessary.

ASCII CHARACTERS

It has been stated above that TECO is used for editing files which consist of ASCII characters. These are the characters which can be entered using a standard terminal; the letters stand for American Standard Code for Information Interchange. There are 128 characters in the full set and each character has a defined code number. For example, "A" is the 65th character in the set and thus has the ASCII code 65 (decimal) or, since these are often expressed in the octal (base eight) system, 101 (octal). In the binary system used in computers (and punched paper tapes) each ASCII character can be represented by a sequence of seven binary digits or "bits" (i.e. Ø's and 1's).

This should not be of much concern to you in using TECO but it is important to realize that the ASCII code is simply a convention; there are other representations which are used in other applications. In particular, the representation of the *single* number 352 used internally by a computer during arithmetic operations is nothing at all like the *three* ASCII characters 3, 5, and 2. To repeat, TECO is designed to edit files consisting of ASCII characters. Generally speaking these are files designed for human readers such as source files, data files and output files.

It is helpful, and common, to set out the list of ASCII characters as shown in the accompanying table. This shows the characters in four groups of thirty-two. The third group, which includes the upper case alphabet, is a convenient reference point. The code numbers here run from 64 to 95 (decimal). The fourth group is rather commonly missing from terminal equipment and includes the lower case alphabet. It can be seen that the code (decimal) for a lower case letter is formed by adding 32 to the code for the corresponding upper case form.

The second group is primarily mathematical and has no direct relationship with the reference group. However, the first group is rather special and the characters in it may be described in two ways. Firstly, they have special names (not all of which are needed in TECO) such as NULL, ESCAPE and TAB. These describe their function, although this may not be immediately obvious to the uninitiated. The second way is to refer to these as "control characters" since they can be transmitted from a terminal by holding down the CONTROL key while striking another character from the third group. Thus NULL is also CONTROL–@ and TAB is CONTROL–I. In effect, the CONTROL key automatically subtracts 64 from the code (decimal) of the other character. Some control characters can only be transmitted in this second form; others, e.g. CARRIAGE RETURN, are generally provided with special keys on the terminal and can thus be given in two forms although it is obviously more convenient to use the special single key under usual circumstances.

Some of the control characters produce a response at the terminal i.e. CARRIAGE RETURN, LINE FEED, TAB, VERTICAL TAB, BACKSPACE (if the terminal is equipped for this) and FORM FEED. The others, on the great majority of current terminals, produce no direct response; they are non-printing characters. This can be checked by switching the terminal to LOCAL (typewriter) mode and striking the control characters. However, and particularly in TECO, it is important to know if a control character has been typed. TECO provides this knowledge through its echoeing convention.

When a terminal is in ON LINE mode (connected to the computer) there is no direct connection between the keyboard (the input device) and the typing mechanism (the output device). The characters entered on the keyboard are transmitted to the computer which processes them and, where appropriate, "echoes" them by sending a signal to the output device. Thus normally if you type a character it will appear on the paper record just as if the terminal had been in LOCAL mode. However, in TECO, all control characters received by the computer are processed specially. Those which have a physical effect on the terminal, such as LINE FEED, are echoed normally except for FORM FEED (which is echoed as a number of LINE FEEDS with a consequent saving in paper) and CARRIAGE RETURN which automatically has a LINE FEED character added after it before both are echoed. There is nothing intrinsically special about this particular group of control characters. The ASCII convention is simply that the signals associated with the (decimal) codes 8 through 13 produce effects on the carriage mechanism rather than on the printing mechanism.

THE ASCII CHARACTER SET

GROUP 1 CONTROL CHARACTERS				GROUP 2			GROUP 3 UPPER CASE			GROUP 4 LOWER CASE		
CONTROL Key and:	Name	Code: Octal	Code: Decimal	Character	Code: Octal	Code: Decimal	Character	Code: Octal	Code: Decimal	Character	Code: Octal	Code: Decimal
@	NULL	ØØØ	Ø	SPACE	Ø4Ø	32	@	1ØØ	64	`	14Ø	96
A		ØØ1	1	!	Ø41	33	A	1Ø1	65	a	141	97
B		ØØ2	2	"	Ø42	34	B	1Ø2	66	b	142	98
C		ØØ3	3	#	Ø43	35	C	1Ø3	67	c	143	99
D		ØØ4	4	$	Ø44	36	D	1Ø4	68	d	144	1ØØ
E		ØØ5	5	%	Ø45	37	E	1Ø5	69	e	145	1Ø1
F		ØØ6	6	&	Ø46	38	F	1Ø6	7Ø	f	146	1Ø2
G	BELL	ØØ7	7	'	Ø47	39	G	1Ø7	71	g	147	1Ø3
H	BACK SPACE	Ø1Ø	8	(	Ø5Ø	4Ø	H	11Ø	72	h	15Ø	1Ø4
I	TAB	Ø11	9	)	Ø51	41	I	111	73	i	151	1Ø5
J	LINE FEED	Ø12	1Ø	*	Ø52	42	J	112	74	j	152	1Ø6
K	VERT TAB	Ø13	11	+	Ø53	43	K	113	75	k	153	1Ø7
L	FORM FEED	014	12	,	Ø54	44	L	114	76	l	154	1Ø8
M	CARR. RTN.	Ø15	13	–	Ø55	45	M	115	77	m	155	1Ø9
N		Ø16	14	.	Ø56	46	N	116	78	n	156	11Ø
O		Ø17	15	/	Ø57	47	O	117	79	o	157	111
P		Ø2Ø	16	Ø	Ø6Ø	48	P	12Ø	8Ø	p	16Ø	112
Q		Ø21	17	1	Ø61	49	Q	121	81	q	161	113
R		Ø22	18	2	Ø62	5Ø	R	122	82	r	162	114
S		Ø23	19	3	Ø63	51	S	123	83	s	163	115
T		Ø24	2Ø	4	Ø64	52	T	124	84	t	164	116
U		Ø25	21	5	Ø65	53	U	125	85	u	165	117
V		Ø26	22	6	Ø66	54	V	126	86	v	166	118
W		Ø27	23	7	Ø67	55	W	127	87	w	167	119
X		Ø3Ø	24	8	Ø7Ø	56	X	13Ø	88	x	17Ø	12Ø
Y		Ø31	25	9	Ø71	57	Y	131	89	y	171	121
Z		Ø32	26	:	Ø72	58	Z	132	9Ø	z	172	122
[	ESCAPE	Ø33	27	;	Ø73	59	[	133	91	{	173	123
\		Ø34	28	<	Ø74	6Ø	\	134	92	|	174	124
]		Ø35	29	=	Ø75	61	]	135	93	}	175	125
↑ or ∧		Ø36	3Ø	>	Ø76	62	↑ or ∧	136	94	~	176	126
← or –		Ø37	31	?	Ø77	63	← or –	137	95	RUBOUT	177	127

The other control characters are echoed by transmitting two characters to the output device. The first is a caret (^), or on some terminals, up-arrow (↑) and the second is the character which is used in combination with the CONTROL key to form the alternate form of the character. Thus CONTROL-A is echoed as ^A (or ↑A). The exception to this is ESCAPE which finds very frequent use in TECO. This has its own key but can be formed as CONTROL-[. For convenience TECO echoes this with the single character $.

The convention followed here is to enclose the echoed representation of a control character within a circle, e.g. (↑E) or ($). This is to avoid confusion with the corresponding single characters. This is not done in actual TECO terminal records, of course, and if there is serious confusion you can suppress TECO's echoeing convention so that control characters are not echoed.

It should be noted that some control characters are transmitted by holding down the CONTROL key while typing a character which is transmitted by holding down the SHIFT key and striking a key. In such cases it is necessary to hold down both the CONTROL and the SHIFT keys while striking the character key. For example, on some terminals the character CONTROL-↑ is entered as CONTROL-SHIFT-N.

THE TEXT (EDITING) BUFFER

When you use TECO your basic aim is to produce a new file. This may be completely new or it may be an altered version of one which already exists. Now you *could* create a new file by writing directly into it (i.e. you type the character "A" on the terminal and the representation of "A" is written directly into the file). This would not be very convenient. It is far better to use the computer's memory (its "core") to store a section of input from the terminal so that you can check it before committing it to your output file.

In the same way if you are altering a file it is convenient to read a section of the file and store it in core before making the changes and outputting the new version of the file. In other words you use the computer memory as a *"buffer"* to "absorb" a quantity of text, or program, which is read from your terminal or existing files so that changes can be made in context and checked before the new file is finally written.

Actually, in TECO there are a number of "buffers"; these are simply areas of core used for temporary storage. The first of these which you will encounter is called the text (or editing) buffer. As the name implies it is here that the text to be changed is temporarily stored; it is empty when you first enter TECO from the system.

On your command, TECO reads your file into the text buffer, it makes the changes which you specify to the material in the text buffer and, again on your command, writes the contents of the text buffer into another file which you specify. You are then left with your old file and a corrected file (with a different name). If you are satisfied with your corrections you can then delete the old file.

If you don't tell the computer to write the contents of the text buffer into a file it will be lost! The text buffer is simply part of the computer core and will be written over by the computer after you have finished with TECO.

Since you can add to the text in the buffer through the terminal keyboard you can use TECO to create an entirely new file. In this case there is clearly no need to read anything into the text buffer and no old file is involved.

The most familiar physically recognizable computer files are undoubtedly decks of punched cards. It is quite likely that you have already done some editing of such card files and have some experience in the basic task of finding the card which is to be altered and then replacing it with a corrected one. In detail there are several ways of looking for that card. You could search for it as the n^{th} card in the deck or as, say, the second card after the first one which has an "X" in the first column.

Alternatively, you might not choose to look for the card itself; you might prefer to look for a particular character or word and change the card which happens to contain the n^{th} occurrence of that character or word. Editing with TECO can follow either of these approaches. You have to specify just what it is that you are seeking and the computer, through TECO, finds the part of the file you wish to change. This beats shuffling through decks of cards.

However, there is an important difference between card editing and TECO editing. The files stored in the TECO text buffer are not arranged in lines (records) like a sequence of cards. They are simply a *sequence of characters.*

You may have seen punched paper tapes used with computers. Each row of holes across the tape represents one character (including such characters as CARRIAGE RETURN, <cr>, and LINE FEED, <lf>). When you punch a tape, using a suitably equipped terminal, the typed record of what you have entered will show a series of separate lines corresponding to your typing of CARRIAGE RETURN LINE FEED combinations. However, the tape will appear as a continuous record of rows of punched holes. Unless you are experienced at reading the punched tape directly you will not know where one line ended and the next began.

The TECO text buffer is similarly arranged. When text has been entered, either through the keyboard of a terminal or from an existing file, it appears to be a continuous sequence of characters. If the buffer is empty it will contain no characters at all; if full (this depends on the size of the computer) it might contain thousands, all in a continuous sequence or "string".

For example, you may have typed on the terminal a section of file which goes like this:

```
    :
CD EF
GHI  J
       KLM
NOP
    :    (etc.)
```

The corresponding section of the text buffer will consist of the following string of characters:

. .CD EF <cr> <lf> GHI J <lf> KLM<cr> <lf> NOP<cr><lf> . . (etc.)

You should note particularly that SPACE is a character and that in TECO LINE FEED is added automatically when you type CARRIAGE RETURN while entering text through a terminal.

THE BUFFER POINTER

The foregoing description of the continuous string of characters stored in the buffer may at first suggest to you that it would be difficult to find the parts of the file you wish to change. In fact this task is made very easy by the invention of a "buffer pointer". This pointer can be placed *between* adjacent characters in the buffer (and before the first and after the last). To repeat, it is placed between, *not* on characters in the buffer.

For example, in the following text the pointer, shown here by an arrow, lies between the C and the D.

. . . ABCD<cr><lf>EFGH<cr><lf>. . .
        ↑

If you move it one place to the right (in TECO this direction of movement is called "forward" or "down") it will lie between the D and the CARRIAGE RETURN (i.e. at the end of the text in the "line" ABCD) and if you move it forward two more places it will be between the LINE FEED and the E (i.e. at the start of the line EFGH<cr><lf>). The LINE FEED rather than the CARRIAGE RETURN is taken to delimit the end of the line.

Actually the pointer is fictional. Internally the computer numbers all the characters in the buffer and your command to move the pointer two places forward is translated by TECO into a form such as "update the internal counter to indicate that the $n+2^{nd}$ rather than the n^{th} character is now being considered". You will appreciate the greater convenience of the pointer concept.

Some versions of TECO which are specifically written for use with Visual Display Unit (V.D.U.) terminals actually show an image of a pointer at the appropriate position in the text on the screen. If you don't have access to such a facility you will still find it helpful to have a mental picture of the pointer just like the image on the V.D.U. screen.

Needless to say, you don't have to know what the internal numbers of the characters are, although you can find out. However it is necessary to remember that pointer position n lies between the n^{th} and $n+1^{st}$ characters; the pointer position before the first character is given the number Ø (zero). The pointer position immediately after the last character in the buffer has the same number as the total number of characters in the buffer. You never have to know what this is because TECO allows you to identify this position with the letter Z (meaning the last character, as in the alphabet).

In fact Z is a TECO *command*, the first mentioned so far. You issue this command, like all TECO commands, by typing the letter Z on the terminal. It makes no difference whether commands are entered in *upper or lower case* form.

The special characteristic of the Z command which allows you to identify it with the number of characters in the buffer is that it is one of a group of commands which is said to "return" a numeric value. That is, apart from any effect on the pointer or text, such commands take on a numeric value which can be used as a "numeric argument" for the following command. This will be illustrated below. As it turns out the Z command does nothing to the pointer or the text; it simply returns the number of characters in the text buffer.

Before any editing can be done it is necessary to locate the point in the text where the changes are to be made. In this chapter only the most direct ways of locating text will be considered. Thus the first "active" commands to be described are those which move the pointer and allow you to confirm where it is located. The first of these is the direct jump. When TECO executes the command nJ (for Jump), where the numeric argument n is a positive integer, or a command (e.g. Z) which returns a value, the pointer will be moved to position n (between the n^{th} and $n+1^{st}$ characters).

In practice you usually only need to use a couple of special cases. The command ZJ moves the pointer to the end of the buffer and ØJ moves it to the start.

The start of the buffer is also identified by the command B (for Beginning) which, like the Z command returns a value (it does *not* move the pointer). The value returned is, however, always zero. It is not obvious why this command has been included in TECO.

In fact the special command J also sends the pointer to the start which illustrates another feature of TECO: there are often a number of ways of doing the same thing. However, most of the time there is a preferred way which is simpler than the others. For example, the integer, n, which is used as the numeric argument of the nJ command, can itself be an integer mathematical expression. That is, (2+3)J, means move the pointer to the position numbered 5, (i.e. between the 5th and 6th characters). Also, just as B and Z refer to special pointer positions the command . (period or full stop, but read as "point" or "pointer") returns the number of the current pointer position. Thus the command (.+12)J would move the pointer forward 12 positions from its current position.

This is quite straight forward but it is a little inconvenient and there is another command, nC (for n Characters) which moves the pointer n positions forward if n is positive and backwards if n is negative. Thus (.+12)J can be replaced by 12C.

It is more surprising that there is a command, nR, (for Reverse) which moves the pointer back n positions if n is positive and forward n positions if n is negative. Thus 12C is equivalent to -12R.

You should be aware of the special meanings for B, Z and . and that the numeric argument can be an expression but you will find that, at the start, you need only know the commands J, ZJ and nC to be able to move the pointer character by character.

If you need to know the value of a numeric argument such as Z or . you can use the "equals" command n= (i.e. n equals) where n is the numeric argument of the = command. This will type the value of n, the numeric argument. Thus the command .= will type the number of characters to the left of the current pointer position. A CARRIAGE RETURN LINE FEED combination is automatically added after the number is typed so that the terminal printer (carriage) is left at the start of a new line after this command is executed.

The concept of numeric expressions has been introduced mainly as background information; you won't need to use expressions in simple TECO editing. However, if you intend to use them you *must* read the later section which describes their use in detail. Expressions in TECO-11 are fairly straightforward but in TECO-1Ø there are (very) unusual features. Only one of these is of immediate concern.

In TECO-1Ø a space in a numeric expression is equivalent to a + operator. It must have seemed a good idea at the time. The (indirect) requirement that this imposes is that you should *not use any SPACE characters* in, or immediately after, numeric expressions. This means that a SPACE must not be used between a numeric argument and the following command. That is, Z J is *not* the same as ZJ. The safest approach is to apply this rule to all forms of TECO. This way you will avoid any risk of confusion in changing between TECO-1Ø and TECO-11.

LINE ORIENTED MOVEMENT COMMANDS

It is often preferable to examine files line by line rather than character by character. TECO can be instructed to operate on this basis in commands which examine the characters one by one until a LINE FEED character is identified. This is taken to mean that the end of one line (and the beginning of the next) has been found. For example if n is positive the command nL (for Lines) moves the pointer to the position immediately after the n^{th} LINE FEED after the current pointer position. That is, if the pointer is currently somewhere in a line of text the command L (or 1L) moves it to the position just before the first character of the next line.

If n is negative the pointer is moved back to the beginning of the n^{th} line before the current line. The command ØL moves it to the start of the line in which the pointer is currently located and -1L (or -L) moves it to the start of the preceding line.

Actually, as well as LINE FEED the characters FORM FEED and VERTICAL TAB, which also advance the output device by a number of lines, and also the end of the buffer, are considered by TECO to define the end of a line.

Thus, with the commands J, ZJ, nC and nL you can move the pointer all over the buffer. The next problem is that of checking the location of the pointer.

TYPE-OUT COMMANDS

With a printing terminal the way to find out where the pointer is located is to type out part of the line in which it is currently located. The command nT (for Type-out) works a bit like nL although it does *not* move the pointer. Thus for positive n, the command nT types out everything between the current position of the pointer up to (and including) the n^{th} LINE FEED after it. The command T (or 1T) types the rest of the current line.

For negative n, the command nT types from the start of the n^{th} preceding line up to the pointer and thus the command -1T (or -T) types the previous line and current line up to the pointer and ØT types from the start of the current line up to the pointer.

If you had a file:

ABCD
EFGH
IJKL

and you didn't know where the pointer was you could give the command T. If the result was that the terminal typed out

FGH

you would know that the pointer is between the E and the F in the second line.

Most of the time you will be content to start typing exactly at the pointer and to stop exactly before the start of a line (n positive). However, you don't have to do this. The command (m,n)T (where m<n and both are positive integers) types everything from the m + 1st through the n^{th} characters in the buffer (without moving the pointer). This is not a general construction; it only works with certain specified commands, one of which is the T command.

This form of the command is usually used in conjunction with commands such as B, Z and . which return a numeric value, for example, (B, .)T (read as "Type out from beginning to the pointer") or (.,Z)T (read as "Type out from pointer to the end of the buffer"). A command such as (37, 1Ø5)T, using explicit numeric values must be very unusual. For example, you may well wish to type out the whole of the text in the buffer (without moving the pointer). This can be done with the command (B,Z) T. In fact this is so common that the numeric argument pair B, Z (or Ø,Z) is given the special symbol H (for wHole buffer) so that if you want the whole buffer typed you give the command HT.

The numbers m and n in the argument pair m,n can themselves be integer expressions. Thus the command ((.-1), (.+1))T would type out the character before and the character after the pointer. Another way you use this would be to type out a specific section of the text. First you would move the pointer to the position immediately before the start of the section to be typed and give the command .= which would type out the number of this pointer position, e.g. 346. Then you could move the pointer to the position immediately after the end of the text to be typed and give the command (346, .)T which (in this example) would have the desired effect.

You should understand that the T command causes just those specified characters from the text buffer to be typed. If these do not include CARRIAGE RETURN and LINE FEED characters then these actions will not be produced on the terminal. Type-out starts wherever the last terminal typing action left the carriage; it does not in general start or end on a new line.

For example the two commands ØT, which types from the start of the line to the pointer, and T, which types the part of the line after the pointer, can be combined to type the whole line. That is, ØTT types the whole line with no indication at all of where the pointer is located within it.

In TECO-11 the command nV (for Verify) types out (n-1) lines on either side of the pointer. This command is equivalent to (1-n)TnT and the command V is thus the same as ØTT.

TECO-11 also provides an automatic form of the Verify command. TECO is set to do this by the "enabling" command nEV (Enable Verify). If n is equal to zero this option is not operative. This is the initial (default) value which applies when TECO is entered. If n changed to a non-zero value with this command a portion of the text around the pointer is typed out automatically before the asterisk prompt character which signifies that TECO is ready to accept more commands. The command EV with no argument returns the current value of the setting (the last n given).

If n is equal to -1 the current line is typed out; if n is in the range 1 to 31 the current line is typed out with a LINE FEED immediately after the pointer position. If n is greater than 31 the current line is typed out with the ASCII character whose (decimal) code is equal to n immediately after the pointer position. You can also choose to type out (m-1) lines on either side of the current line as well with the command (m*256+n)EV where n is as described above.

This keeps you informed of where the pointer is after every command execution and is the next best thing to V.D.U. version of TECO. However, you really need a fast terminal to use it; if you use a ten character per second printing terminal you will probably find it too slow (and wasteful of paper) since with only a little practice you will find that in many cases type-out is redundant. It is very useful while you are learning.

You don't need to remember the ASCII codes. The command ↑↑x (CONTROL-CARET, or CONTROL-UPARROW, x or, on some terminals, CONTROL-SHIFT-N x) returns the ASCII code of the character x. The characters commonly used with this command to show the pointer position are LINE FEED, PERIOD (FULL STOP), read as "point" and EXCLAMATION POINT. Thus to set TECO to type out the current line with an exclamation point (exclamation mark) after the pointer position the command ↑↑ !EV can be given.

TECO-11 provides for the reverse of the ↑↑x command. The command:

n ↑T

types out the character whose (decimal) ASCII code is n.

The FORM FEED command can sometimes usefully be given before a type-out command. This is simply the form feed character (CONTROL-L). In TECO it is echoed as a few line feed characters to avoid needless waste and when you type FORM FEED the paper advances a few lines. The advantage is that when this command is executed the same action occurs. If you give the three commands:

↑L

(note the line feed actions produced)

HT

the whole buffer will be typed out clearly separated from the other typing on the page. This gives you a clean copy of the text. It is shown later in this chapter how you can enter the FORM FEED command without having the corresponding action echoed.

If you add the commands nT and HT to J, ZC, nC and nL you will be able to move the pointer around, check where it is and type out the contents of the buffer. However, you won't yet be able to do anything to the contents of the buffer.

TEXT DELETION

The essential things to do with the text in the buffer are to delete unwanted portions and to add new sections. The first of these operations can be done with the command nD (for Delete) which for positive n deletes the n characters immediately following the pointer and for negative n deletes the n preceding characters. Deletion is exactly what happens; the n characters are removed and the string of text in the buffer is closed up so that the pointer is left between the two characters either side of the deletion. Remember, a SPACE is an ASCII character, just like a letter.

TECO includes a line-oriented deletion command nK (for Kill) which for positive n deletes all characters from the pointer up to and including the n^{th} following line feed character and for negative n deletes from the start of the n^{th} preceding line up to the pointer. The line in which the pointer is currently located can be deleted in two ways. If the pointer is already at the start of the line the command K (or 1K) will do the job. If the pointer is not at the start of the line you could first give the command ØL followed by the command K. Alternatively you could give the command ØK which deletes from the start of the current line to the pointer and then give the command K which completes deletion from the pointer to the end of the line.

The K command is one which takes the alternate form (m,n)K and thus the command HK will delete all the text in the buffer. In TECO-11 the command (m,n)D is allowed as an alternative to (m,n)K. The (m,n)K and (m,n)D commands automatically include the command mJ; the pointer is always left at the deletion.

COMMAND FORMAT AND THE ESCAPE KEY

Now that you know how to move the pointer around the buffer and how to type out its contents and delete unwanted sections only one more command is required to complete the basic set of commands for changing the contents of the text buffer. This is the command to insert new characters. To do this you will have to type (on the terminal) the string of characters to be inserted. This however, raises a number of points which must be considered before the details of text insertion are given.

When you type a command on the terminal keyboard it is not, in general, executed immediately. This has been implied in examples already shown, such as ZJ, in which a number of commands have been combined. The compound command is not executed by TECO until you indicate that it is complete. In many other system programs execution starts when you type a CARRIAGE RETURN. This is not a suitable character to use in TECO because when you come to insert text into the buffer you will wish to include CARRIAGE RETURN characters and you will not want this to initiate execution.

The solution used in TECO is to use the special ASCII character called ESCAPE. It is also called ALTMODE or PREFIX on some terminals. It generally has its own key (marked ESC, ALT or PRE) but can also be entered as CONTROL-[(CONTROL-SHIFT-K). The function of this command, which is interpreted immediately it is entered, is to *alter* the current *mode* of interpreting characters entered from the terminal. In other words it signals the computer to "escape" from the current mode of interpretation. The ESCAPE character is not a printing character on most terminals but TECO is set to echo a $ character whenever ESCAPE is entered. In these notes this echoed symbol is written Ⓢ to distinguish it from the actual $ sign.

A single ESCAPE signifies that a command is complete. You could type a Ⓢ after each complete command (e.g. *J Ⓢ 3T Ⓢ Ⓢ) but in fact this is unnecessary and generally not done since there is no change in interpretation involved. If you type Ⓢ after a numeric command such as Z, it is treated as complete and does *not* act as a numeric argument. The main use of the single ESCAPE will be illustrated in connection with insertion of text.

Double ESCAPE, that is typing ESCAPE immediately followed by another ESCAPE, signals that the whole group of commands is to be executed. Execution starts immediately the second ESCAPE is typed. The double ESCAPE resets the terminal to the start of a new line so that the first type-out command will appear on a new line. This is not necessarily true of subsequent type-out commands. The asterisk prompt character which signals that the previous commands have been executed and that TECO is ready to accept more commands does not necessarily appear on the start of a new line; it is simply typed at the position of the terminal carriage at the end of the last type-out operation.

For example, the text buffer contains:

ABCDEF<cr><lf>GHIJKL<cr><lf>

The following terminal record is produced:

*JLTJ3CØT Ⓢ Ⓢ
GHIJKL
ABC*

TEXT INSERTION

The description of the ESCAPE character should make the operation of the insertion command reasonably clear. This command has the form

I . . . text . . . Ⓢ (I stands for Insertion)

and its effect is to insert the string of characters denoted . . . text . . . into the text buffer immediately to the left of the pointer so that the pointer is located immediately *after* the last character inserted. The string . . . text . . . is referred to as a *text*, as distinct from a numeric, *argument.* In this case it is the argument of the I command.

When the I is encountered during the execution of a command string TECO changes its mode of interpretation so following characters are interpreted as text to be inserted and not as commands. A "T" is simply the letter T and does not stand for the type-out command. This mode continues until an ESCAPE character is encountered in the command string. When this is met the mode of interpretation reverts to the normal condition in which characters represent commands.

With the exception of a few special characters which will be described later, all characters lying between the I and the (Ⓢ) are inserted into the text in the buffer. This includes SPACE, CARRIAGE RETURN and LINE FEED characters which should, therefore, only be used when it is intended that they should be entered into the buffer.

As an example of this command, if the pointer is currently in the line ABCDE<cr><lf> as shown:

*T ⒮⒮
CDE
*

The following command:

*IXYZ ⒮ØTT ⒮⒮ will produce:

ABXYZCDE
*

Since all insertions take place immediately to the left of the pointer, insertion commands can be written in series with no difficulty. For example,
*IA ⒮ IB ⒮ IC ⒮⒮ has the same effect as
*IABC ⒮⒮

With TECO-1Ø alphabetic characters are inserted in exactly the same case (i.e. upper or lower) as they are typed. However TECO-11 makes all insertions into the text in upper case form regardless of how they are typed on the terminal. There is, however, a way of inserting lower case letters which will be described later.

If you are inserting a large block of text it is a good idea to limit insertions to about ten to fifteen lines at a time. As has been shown, there is no problem in doing so and this practice can avoid a couple of things which sometimes cause trouble (such as overfilling the command buffer). These are unlikely events but it is not much fun to have to retype a very long command. You will find that it pays to be fairly conservative when learning TECO.

You must expect to make (at least) a few typing errors when giving long commands. These can be corrected, provided you see them, before the command string is executed by using the command editing techniques which will be explained shortly.

COMMON INSERTION ERRORS

One of the commonest errors made by beginners occurs when inserting a new line. To do this the pointer is placed between the <lf> character at the end of the line before the insertion, that is, before the first character of the line which will follow the insertion. If you forget to type <cr> at the end of a string of characters which forms the new line you will simply add these at the start of the line which was meant to follow the inserted line. If you have the following text in the buffer with the pointer between the first LINE FEED and the D:

ABC<cr><lf>DEF<cr><lf>

and you wish to insert the line XYZ between the two lines you should give the command

*IXYZ<cr> (<lf> is added automatically)
($)

If by mistake you type IXYZ ($) the command HT will reveal:

ABC
XYZDEF

This also reinforces the recommendation that you should frequently use the T command to check that the changes you desired were actually carried out.

Commands which are followed by a text argument string (I is the only one mentioned so far) must *always* be terminated with ($) . The command

*IABC ($) ØT ($)($)

will insert ABC and type from the start of the line to the pointer. However, the command

*IABCØT ($)($)

will insert ABCØT and not type anything. Almost every TECO user will make such a mistake at some time.

A potentially more disastrous error results if you forget to precede the string to be inserted with the letter I. This is quite easy to do. The result is that the characters to be inserted are treated as commands and since almost every character on the keyboard is a command of some kind (only a few have been mentioned so far) various unintended operations will be performed until an invalid command is encountered. An error recovery technique which will allow this mistake to be corrected, if you recognize immediately that it has occurred, will be described in a later chapter.

Only the user can guard against these errors.

COMMAND EDITING

As you have probably realized, when you type commands on the terminal they are stored in a buffer (the command buffer) until the double ESCAPE starts execution. While awaiting execution these commands can themselves be edited. After all, you will inevitably strike the wrong key sometime. In fact it is most *important* that you develop the rigid automatic habit of checking (not just looking at) the command string before you strike that second ESCAPE. The few seconds required are nothing in comparison with the time and effort involved in recovering from unintended commands.

The simplest command editing character is the RUBOUT key. When you strike this key the last character entered through the terminal is deleted from the command buffer and TECO echoes the deleted character on the terminal. If you strike the RUBOUT key again the character which is now the last in the command buffer will be deleted and echoed. You can delete the entire command by repeatedly striking the RUBOUT key. If you type:

*J2L3T and then type RUBOUT twice, the terminal record will show:
*J2L3TT3 and the command in the buffer will be:
*J2L

You should note that double ESCAPE is a special combination which has its special significance of signalling the start of execution only when entered as two consecutive characters from the terminal. You could place two consecutive ESCAPES in the command buffer (e.g. by typing one ESCAPE followed by an arbitrary character which you then delete, and then a second ESCAPE) but this will not start execution. In TECO-11 it has no significance at all. However in TECO-1Ø, when this combination is reached during the eventual execution and any commands which may have been entered after it will not be executed. For example, the command:

*J3CØT ($) x RUBOUT x ($) T ($)($) , where x is the arbitrary (deleted) character, will type only the first three characters in the text buffer. The command T is not executed.

The CONTROL-U (↑U) character provides a more powerful deletion function than the RUBOUT key. This character deletes everything in the command buffer back to (but *not* including) the last CARRIAGE RETURN LINE FEED combination. If you want to delete back past this point you must give a RUBOUT to delete the LINE FEED. Another ↑U will then extend the deletion back to the CARRIAGE RETURN LINE FEED combination before that. Changes to the command buffer can only be made by successive deletions working back from the last character entered; there is no direct way of making a change in the middle of the command without going through this process.

You *can* delete the whole command string by typing enough ↑U characters (with RUBOUTS as required). Complete deletion is signalled when TECO types another prompt asterisk on the terminal. The ↑U character does not echo the deleted characters.

A command string which has been extensively altered can be very hard to read. The echoeing of deleted characters can be quite confusing. However, the combination ↑G SPACE (↑G gives an audible signal on the terminal) causes TECO to type out on the terminal the current version of the last line of the command string. This is a "clean" version with no echoed characters from the RUBOUTs you may have used. Everything in the command buffer from the last CARRIAGE RETURN LINE FEED combination to the end is typed out.

In TECO-11 (only) the current version of the whole command buffer back to the asterisk prompt character can be typed out with the ↑G* combination.

If you have managed to make a complete mess of the command string and consider that it would be better to start again the easiest thing is to type the combination ↑G↑G . This aborts the entire command and you are free to start afresh. This is not exactly the same as actually deleting everything in the command buffer but in this context it has the same effect.

It should be noted that these editing characters are not themselves included in the command string. Neither do they have any effect on the pointer or the text buffer.

MORE ON COMMAND FORMAT

Commands can be written as a long continuous string. This is not, however, a good practice; some formatting of command strings is very desirable.

Except in text argument strings where every character, except control characters, is included as text, the three characters CARRIAGE RETURN, LINE FEED and SPACE are ignored (with an exception described below) in the command string. Thus individual commands can be separated by blanks and command strings can extend over several lines. As a rough guide, write your commands in groups as you would say them. For example, "Jump to start, type a line, move on two lines and type it" would be typed as JT2LT<space>. If the command is too long to say comfortably it is too long to write without a break. As a guide you should probably have a space after every three to six characters in a control string (except for text arguments). Until you gain experience you should not put too many commands in one command string. If, with experience, you write longer command strings use a new line after every, say, five or six such groups.

The command string

** JT2LTZJ-LTJ3LKT $$*

is easier to understand in the following form

**JT2LT ZJ-LT J3LKT $$*

This is meant as a reasonable guide; the aim is to produce an easily checked command. Don't separate numeric or text arguments from their commands!

Some computer systems are arranged so that if you type more characters than will fit on one line of the terminal a local CARRIAGE RETURN LINE FEED action is produced on the terminal but is not transmitted. You can quickly check if yours works this way. If it does not you must strike LINE FEED to move to a new line. As mentioned above this will only be significant if it is done in a text argument. If you have to do this and you don't want the CARRIAGE RETURN and LINE FEED to be included in the text you simply strike RUBOUT twice to remove the unwanted characters.

Rather unfortunately (↑I) (i.e. TAB) has a special command function and cannot be used for formatting. TAB is an insertion command:

<tab> . . . text . . . ($) is equivalent to I<tab> . . . text . . . ($);

it inserts a TAB and . . . text . . . to the left of the pointer. Some users find it handy but it is recommended that it should not be used, particularly if you intend to progress to programmed TECO editing.

It has already been stressed that SPACE characters are not to be used within or immediately after numeric expression or arguments. In contrast, ordinary parentheses, (and), may be used freely with their usual meaning in numeric expressions. Their use is strongly recommended in any situation involving possible ambiguity, particularly in expressions in which negation is specified.

The parentheses which are used with the compound numeric argument, as in (m,n)T are strictly optional, though the comma is essential. It is, however, a good idea to use them.

COMMAND REPETITION

A group of commands may be executed repeatedly by enclosing the group within a pair of angle brackets. The opening angle bracket may or may not be preceded by a numeric argument (which may be an expression). A positive argument preceding the opening angle bracket indicates the number of repetitions. For example the command 5<2CD> will delete every third character after the pointer until five have been deleted (satisfy yourself that this is so). This is a trivial example of the use of angle brackets but in fact they are a powerful feature. Commands enclosed in angle brackets can be nested just as arithmetic expressions can be nested in parentheses with the maximum depth of nesting being in the range 1Ø to 2Ø levels depending on the system. The closing angle bracket terminates the enclosed commands; a numeric argument cannot be passed across it to the next command in the string.

The second option with angle brackets is to have the preceding argument zero or negative. In this case all the bracketed commands are skipped altogether and execution continues with the first command after the corresponding closing angle bracket. This is the first command encountered which indicates TECO's capacity for conditional execution of commands. These powers are described fully in the section on programmed TECO editing.

If any of the commands described so far are enclosed in angle brackets *without* a preceding numeric argument their execution will be repeated indefinitely. To stop this process you will have to abort the execution with the appropriate methods described below. Nevertheless this form of repetition has an important place in more advanced editing.

INVALID COMMANDS

The command -1ØJ is obviously invalid since the pointer positions are numbered from Ø to Z. The command J(-1Ø)C is also invalid and so is ZJ1ØC.

TECO executes all commands in the command string until an invalid command, such as these, is encountered. It does not attempt to execute the invalid command but instead types out a question mark followed by a three letter error code and a one-line explanation of the error. Two commands are available to assist in explaining the error. Expanded explanations are listed in the appropriate manuals.

If, after an error message is received, the command ? (Question) is given as the *very next* command the source of the error can be identified. Typing ? (no ⓈⓈ required) causes TECO to type that section of the command string which preceded the invalid command (these were executed) and that part of the invalid command up to and including the character at which the error was detected and finally to type out another question mark. In TECO-1Ø the maximum number of characters typed is ten.

For example, typing ? after the command
*JT2LT5LTZJ1ØCT ⓈⓈ had produced the response:
?POP ATTEMPT TO MOVE POINTER OFF PAGE (or POINTER OFF PAGE)
would cause the following type-out:
*?LT5LTZJ1ØC? (TECO-1Ø)
*

This is very useful because it allows you to find out just how much of the command string has been executed.

The procedure described above is that followed by TECO in default of any overriding instructions. The command 1EH (for Enable Help) sets TECO so that only the question mark and three letter code is typed after an invalid command has been encountered. The command 2EH sets TECO to type the code and brief explanation. This is the default option. The alternate default command is ØEH (read as no special action).

The command EH with no numeric argument can be used to find the current error explanation procedure. The command takes on this value and thus becomes a special numeric argument like B, . or Z. Its value can be typed out with the = command, *EH = ⓈⓈ .

In TECO-1Ø the expanded explanations are also stored in a file which is accessible to TECO. The command 3EH sets TECO to type the code, the one-line explanation and the detailed explanation. If any other EH option is current the detailed explanation will be typed if a slash, /, is typed after the error is detected and before any other command (except ?) is given. Both / and ? can be used (in either order) after the error is reported, provided that no other command is given.

In TECO-11 there is no accessible file of expanded error explanations. The command 3EH sets TECO to type out the code and the one-line explanation and in addition to type out the command up to and including the command which caused the execution to stop, just as if a ? had been typed.

If you do get an error message be sure to take note of it! You can get into strife if you continue to give new commands after ignoring an error message. As will be described later, you can recover fairly simply from some invalid commands, but *only* if you recognize the error as soon as it occurs.

A point to note is that if an illegal J, C, R, or D command is given TECO can immediately tell that it is illegal because it knows the number of characters in the buffer. However a line-oriented command is different because TECO doesn't know how many <lf> characters are present. If you command 1ØL and the pointer was in, say, the second last line the pointer will move to the end of the buffer in its search for 1Ø<lf> characters but will not move past the last position. This command is *not invalid* however the pointer will have moved only to the end of the buffer. Similarly, the pointer will be moved to the start of the buffer by an excessively large negative argument for the L command. You should make frequent use of the T command to verify the pointer position.

The worst kind of mistake is an unintended command which is *not* invalid but doesn't do what you want. The only way to discover such errors is to check frequently what you have actually done. As has been mentioned, forgetting the I at the start of an insertion command causes many problems but at least execution usually ends with an invalid command which warns you that something has gone wrong. You get no warning if you enter two adjacent numeric commands, forgetting to separate them. For example, (.Z) instead of (.,Z) or (5.) instead of (5+.). In this situation TECO ignores all but the last command in the string of adjacent numeric commands. Thus (5.) is taken to be simply (.). A preceding numeric argument before the string continues to apply. This is not invalid; there is a call for this construction. However, if you did not intend this you are in trouble.

STOPPING

There are three main reasons for wanting to stop what you are doing when using the TECO program. The first is the completion of the editing task; you will wish to return to the system program. The second is that you have recognized that your command is taking a very long time to execute and is probably in an infinite iteration; you will wish to abort the execution. The third is that you have given a command to type out an unnecessarily long section of text; you will wish to abort the type-out.

The basic way you leave TECO is via the ↑C character. However the exact mode of operation is dependent on the particular combination of the TECO version and the operating system. In TECO-1Ø the monitor recognizes ↑C characters as they are entered; ↑C is a monitor command, not a TECO-1Ø command. A single ↑C results in immediate return to the monitor except when a command is being executed. In the latter case double ↑C will abort the execution and return control to the monitor. You can reenter your TECO job from the monitor as described below. Aborting execution is an abrupt procedure. It is up to the user to find out how much of the command was executed.

There is a TECO-1Ø command which returns control to the monitor. This is the ↑Z character. This command is automatically followed by a CARRIAGE RETURN LINE FEED combination when entered from the terminal. It does not have an immediate effect in the way ↑C does; instead, it is entered in the command string so that when execution of the string reaches the ↑Z control is returned to the monitor. The ↑C and ↑Z characters have effects on input and output which will be described in the appropriate section.

In TECO-11 ↑C is a command; in standard systems ↑C within TECO is not recognized as a system command with an immediate function. Thus a single ↑C character can be included in the command string (like the ↑Z in TECO-1Ø) to cause exit from TECO when command execution reaches this point. CONTROL-Z is a completely different command in TECO-11.

In TECO-11 you can abort execution by typing double ↑C . However this does not result in exit from TECO; instead it produces the message

?XAB EXECUTION ABORTED
*

and leaves TECO-11 ready for more commands. Again, it is up to the user to determine just what the execution had achieved before it was aborted.

It is advisable to consult the TECO reference manuals for details of the operation of ↑C . These differ from system to system. In both DECsystem-1Ø and RT-11 systems you can return to the monitor to perform simple system commands (e.g. ASSIGN) which do not call other programs into the core over the TECO core areas and then return to the TECO job with no effect on the contents of the editing buffer. This is achieved with the monitor command .REENTER. The command buffer is, however, reinitialized and any commands which were not executed before the return to the monitor are, in general, lost. They can be accessed by a command described later.

The contents of the buffer will be lost when you finally leave TECO by these commands. Saving the contents by outputting to more permanent files is covered in a later chapter.

If you wish to abort type-out you simply strike the ↑O character. This stops the typing without affecting the execution of the command in other respects. The completion of the execution is signalled as usual with the asterisk prompt character which appears on the start of a new line. Occasionally the ↑O character will suppress the typing of this asterisk, in which case it can be recovered by typing ↑U .The ↑O character only works this way when entered during execution of a command string. Sometimes a command string may include a number of type-out commands. The ↑O command cancels all type-out in that string (after it is given); it does not apply only to the type-out command current when it was given. In TECO-11 the ↑O command can be altered so that it applies only to the type-out command current when it is given so that subsequent type-out commands in the same command string. The details of the command to do this will be explained later but it can be achieved with the command:

(ET#16)ET and can be undone with the command:

(ET#16-16)ET

This is an example of the Enable Terminal terminal control command.

THE CARET (UP-ARROW) COMMAND

TECO allows an interesting extension of its echoing convention for control characters. Control characters which are TECO commands can be entered in a command string in an alternate two-character form by typing CARET (or UP-ARROW) followed by the character which would normally be struck while holding down the CONTROL key. The command string doesn't actually contain the control character when it is entered this way; it contains the two characters as typed. However, during execution of the command string the CARET command signifies that the following character is to be interpreted as if it had been entered while the CONTROL key was held down, i.e. as the corresponding control character.

As an example, the FORM FEED command described previously could be entered as CARET/L (the alternative form is CONTROL-L). If this form is used no form- or line-feed actions will be echoed at the terminal when the command is entered.

According to one school of thought the CARET form should be used for *all* TECO commands and the CONTROL characters should only be used where they have an *immediate* function (e.g. ↑U). This is not a bad idea but it can only be applied fully in TECO-11. You should choose the convention that suits you; the CONTROL form is used here because both TECO-1Ø and TECO-11 are being described. A disadvantage of mixing conventions is that it complicates command string editing. CONTROL-Z is one character and CARET/Z is two yet both are echoed exactly alike by TECO.

In non-standard systems using TECO-11 in which ↑C has immediate system command function the TECO-11 command ↑C must be entered in the CARET/C form.

You can actually enter an undefined command combination such as CARET/U in the command string. However execution will cease and an error message will be given when it is encountered; it is not a valid TECO command. In TECO-1Ø the character C, which is normally a valid command, is not defined as a command if it follows a CARET.

SPECIAL CHARACTERS

A number of special-function characters have been described. These are ESCAPE, RUBOUT, ↑U and ↑O (and ↑C in TECO-1Ø). In addition, the combinations ↑G <space>, ↑G ↑G and ↑C ↑C also have special functions. The particular system in operation may also have other reserved characters.

Obviously you can't insert these into the text buffer using the I . . . text . . . $ command. At this stage it is most unlikely that you would want to. However, TECO does possess the power to handle these as if they were ordinary characters and this is described in later sections.

SUMMARY OF CHAPTER ONE

This chapter has described the fundamental commands of TECO: how to enter and leave TECO; how to move the pointer to the part of the text which is to be changed and verify this position with the type-out command; how to delete unwanted text, insert new strings of characters and verify the change with the type-out command. The commands which will be described in later chapters perform some of these operations more conveniently but the basic sequence of editing operations remains the same.

It has been explained that some commands may take numeric arguments and some take text arguments (delimited by the command letter and an ESCAPE character). These commands may be given in a long command string which may be corrected or altered before its execution is commanded by typing double (Ⓢ). Some commands move the pointer, others have effects on the text. Some return numeric values while yet others control the execution of the command string.

If one thing is to be stressed it is the need to be CAREFUL. Read your command string before you type double ESCAPE. Take notice of any error messages.

FIRST TECO EXERCISE

The best way to learn is to practise. You have now been given sufficient information to create and edit a string of text. After the appropriate system command e.g.

```
.R TECO <cr>
*
```

The computer responds with an asterisk. You should then insert a string of text and then move the pointer through it, typing out to verify the pointer's position. You should then modify it by adding and deleting strings of characters. You need only use the commands

J, ZJ, nC, nL, nT, HT, nD, nK, HK (careful!) and I . . . text . . . (Ⓢ)

although you may like to try using such forms as (m,n)T, compound arguments such as (3+.) and the repeated operation using angle brackets.

It is an excellent learning exercise to edit using this restricted set of commands. You should master these although more advanced forms given later can be used in place of some of them.

If you are using TECO-11 your first command should be:

```
*(↑↑)! EV (Ⓢ)(Ⓢ)
!
*
```

which causes TECO to type out the current line automatically with an exclamation mark where the pointer is. The example here shows that the buffer is empty. If you are using TECO-1Ø you should add the command ØT to the end of every command string (just before the (Ⓢ)(Ⓢ)). This will type out the current line up to the pointer. It is better to use ØT than T because the former types out an insertion made to the left of the pointer.

Don't forget the I (for Insert) at the start of your insertion text argument! You only need a few lines in the buffer for this exercise. Finally, even if it is stating the obvious, read this chapter carefully before you attempt the exercise and write down what you intend to do before you get to the terminal.

2. ADDITIONAL TEXT MODIFICATION COMMANDS

The set of commands already given is sufficient for basic editing. However there are some other very convenient commands which you will appreciate, particularly after some practice with the basic set.

STRING SEARCH

TECO can be instructed to search for a specified string of characters in the text buffer and, if successful, to reposition the pointer at the end of this string. This can save you the trouble of, say, counting the number of characters from the start of the line to the end of the string and then giving the C command. You may have discovered that this is not at all convenient when you are not sure if spaces are actually <space> characters or have been entered with CONTROL-I i.e., <tab> characters.

The S (for Search) command has the form:

nS . . . text . . . (Ⓢ)

where . . . text . . . represents a string of ASCII characters and n is a positive integer (assumed to be equal to 1, if not specified). TECO searches the text which lies *after* the current pointer position for the n^{th} occurrence of the specified string, and, if the search is successful, repositions the pointer after the last character. In TECO-11 the search string can contain up to 128 characters which are to be matched in the text; in TECO-1Ø the limit is 36 but this is quite enough in most editing procedures.

If the string does not occur n times the pointer is moved to *start* of the buffer and an error message is typed. However, if the command was given *within* an iteration (i.e. the string containing the command is enclosed within angle brackets) failure is not an error; the pointer is still moved to the start of the buffer. In TECO-11 a message is typed:

% SEARCH FAIL IN ITER

but execution continues.

The S command is very convenient but you have to ensure that the search string is sufficiently specific. If you wish to place the pointer at the end of the word THOUSAND (which only occurs once, say) you could give the command JSTHOUSAND (Ⓢ) (You should note the advisability of the J command). It would probably be sufficient to use JSUSAND (Ⓢ), since "USAND" doesn't appear in many words. The commands SAND (Ⓢ), SND (Ⓢ) or SD (Ⓢ) are progressively less specific and would be useless for locating "THOUSAND" in a large text. However, even the last of these might be quite sufficient if, for example, you had typed the part of the line after the pointer and could see that "THOUSAND" occured and was the first word which ended with the letter D.

You should use the T commands to verify that you have located the string you wanted.

THE FUNCTIONAL STRING SEARCH

This command, abbreviated FS (for Functional Search), is an extension of the S command. It has the form:

nFS . . . text1 . . . (Ⓢ) . . . text2 . . . (Ⓢ)

where n is a positive integer, and its effect is to search for the n^{th} occurrence of . . . text1 . . . in the buffer after the present pointer position, delete this string, replace it with . . . text2 . . . and leave the pointer positioned immediately after the newly inserted string.

If . . . text1 . . . is not found in the portion of the text in the buffer after the present pointer position, the search fails, no deletion or replacement occurs, the pointer is moved to the start of the buffer and unless the command was used within an iteration, an error message is typed.

The FS command takes the place of the command string: S . . . text1 . . . (of m characters) (Ⓢ) -mDI . . . text2 . . . (Ⓢ). It executes three commands in one (and also counts the number of characters in string 1). All multiple function commands must be treated carefully; with a series of single function commands you are more conscious of what each command will achieve.

Until you gain experience be wary of using this command in an iteration loop. You can easily make an unknown number of unintended changes which you might not be able to check without retyping the file and examining it carefully.

You can use this command to find and delete strings without replacement. However the command FS . . . text . . . (Ⓢ)(Ⓢ) which accomplishes this is executed immediately because of the double (Ⓢ). If you want to include it in a longer string of commands you could use FS . . . text . . . (Ⓢ) x (Ⓢ) -D to change . . . text . . . to an arbitrary character, x, which is then deleted. An alternative to this procedure will be described in a subsequent section.

TECO-11 allows the character (↑R) as an alternative to the two characters FS in this command. This is a mnemonic for Replace. The FS form will be used here.

TECO-11 allows the second part of the FS command to be executed separately as the FR (Functional Replacement) command. This has the form:

FR . . . text . . . (Ⓢ)

and is equivalent to the command sequence:

-nDI . . . text . . . (Ⓢ)

in that n characters (the length of the last insertion) to the left of the pointer are deleted and replaced with . . . text . . . as specified in the argument. This command is sueful if you execute a search successfully and then realize that you should have given an FS command. This only holds if the pointer is still at the end of the matched string and no other search commands have been executed.

SPECIAL TECO-11 SEARCHES

TECO-11 provides for a number of variant forms of the Search (S) and Functional Search (FS) commands. The commands

-nS . . . text . . . (Ⓢ) and -nFS . . . text . . . (Ⓢ)

which involve a negative argument are similar to the positive argument commands except that the searches proceed in the opposite direction, i.e. from the pointer back towards the start of the buffer. If the pointer is initially in, or at the start of, the specified text string this is counted as the first occurrence. If the search succeeds the pointer is moved to the position immediately after the specified occurrence and if it fails the pointer is moved to the start of the buffer and an error message is printed (same as for positive argument search).

The bounded search features are more significantly different. The commands

(m,n)S . . . text . . . (Ⓢ) and (m,n)FS . . . text . . . (Ⓢ)

in which both m and n can be positive or negative and, in addition, m can be zero, execute the searches as defined by the sign of n but fail if the pointer has to be moved more than ABS(m)-ABS(m)-1 positions to obtain a match. If the specified occurrence of the string is found within the specified extent of text the pointer is moved to the end of that string (this may involve a move of more than ABS(m)-1 positions). If the bounded search fails the pointer is *not* moved. The special case of m equal to zero is equivalent to an nS or nFS search except that the pointer is not moved if the search fails.

The command:

(1, 1)S . . . text . . . (Ⓢ)

has a very useful function as a comparison rather than a search command. If the string of characters immediately following the pointer matches the specified text then the pointer is moved to the end of this string; if not the pointer is not moved.

TECO-11 also provides a convenient way of specifying a bounded search in terms of a number of lines rather than characters. The command n (↑Q) allows this. The argument n is taken to be a specified number of line terminator characters after (before, if negative) the current pointer position and the value returned by the (↑Q) command is the number of characters between the pointer and this line terminator. A bounded search can thus be limited to m lines (rather than m characters) by entering it as

(m (↑Q), n)S . . . text . . . ($)

If (↑Q) is not available to TECO in your system use CARET/Q.

These commands are quite useful, particularly in advanced editing. However similar results can be obtained (though not as conveniently) in other ways. Users of TECO-1Ø manage well enough without them.

AUTOMATIC SEARCH TYPE-OUT

The advisability of frequent checking of the pointer position has been stressed. Inexperienced users in particular may find the automatic search type-out option useful.

This option is enabled with the nES (Enable Search type-out) command. This is very like the nEV command which has been described previously but applies in both TECO-1Ø and TECO-11. The difference is that the type-out only occurs after a successful search and not after every command string execution. The various forms of this command are exactly as described for the nEV command except that the option of printing the surrounding lines is not available in TECO-1Ø. To summarize, the default setting is zero (no effect) and if n=-1 the current line is typed. If n is in the range 1–31 a LINE FEED is given after the pointer and if n is greater than 31 the ASCII character whose decimal code is n is given after the pointer position. In TECO-11 if n=(m*256+n) (m-1) lines either side of the current line are also printed with the pointer position defined by n in the expression. The command (↑↑)x can be used in conjunction with this command.

Thus if a line contains the characters ABCDEFG<cr><lf> and the command SDEF ($) is given to search it for the string DEF, the following results can be obtained:

* ØES SCDE ($)($)
* (no typeout)
* -1ES SCDE ($)($)
ABCDEFG (line typed)
* 1ES SCDE ($)($)
ABCDE
FG (line feed at pointer position)
* 32ES SCDE ($)($)
ABCDE FG (space at pointer position)

One of these options may suit you. Once this command is given it applies throughout the TECO job or until it is changed by another ES command.

Automatic type-out never occurs if the search command is in an iteration loop (enclosed within angle brackets).

The command ES with no numeric argument is used to find the ES option currently in force. The command returns the value of the argument n which was given in the last nES command. That is, the ES command, like the Z command, is a numeric argument and its value can be typed with the equals command. Thus continuing the previous example:

* ES = ($)($)
32
*

SEMICOLON – MODIFIED SEARCHES

The search commands may validly be placed inside angle brackets to form an iteration loop without necessarily having a numerical argument to precede the opening bracket. Provided that the command string does not allow the same occurrence of the string to be found repeatedly (as it would be in the command <SABC(Ⓢ)ØL>) the iteration must come to an end eventually because the search string can only occur a finite number of times.

The command J<FSX Ⓢ Y Ⓢ> would certainly change every X in the buffer to a Y but it would not end when the last search for an X failed. This is not an error condition (because the FS command is enclosed within angle brackets), but the pointer would be moved to the start of the buffer and execution would continue indefinitely. This can be avoided with the semicolon command.

In the present context (the more general function of this command is described later), the semicolon command is used to modify a search command in a command string which is *enclosed* in angle brackets. This command is given immediately after the search command to which it applies and its effect is to modify the action taken by TECO if that search command should prove unsuccessful.

If the search which has just been completed was successful the semicolon has no effect at all. If, however, the last search was unsuccessful all commands following the semicolon in the string which is enclosed by the angle brackets are ignored and the first command after the right angle bracket (which terminates the iteration loop containing the semicolon) is then executed. The movement of the pointer is the same as for normal searches.

This procedure provides the correct way to change every X to a Y. For example, you may wish to make this change, return the pointer to the start and then type the whole text. This can be achieved by the command J<FSX Ⓢ Y Ⓢ;>JHT. When the last X is found it will be changed to a Y and, since the last search was successful the enclosed search command is repeated. This time the end of the buffer will be reached and no X will be found and, since the search has failed, the next command executed will be J followed by HT.

The same could be achieved by the command J<SX Ⓢ;-DIY Ⓢ>JHT. Note that the semicolon is placed immediately after the *search* command which it modifies.

The semicolon command is used only within angle brackets. It provides the correct way of terminating an indefinite search loop; it also suppresses search failure warnings.

As another example, the following command types out every line in which the word READ occurs at least once J<SREAD Ⓢ; ØLTL>. The final L inside the loop advances the pointer to the next line after the first occurrence of READ in the line has resulted in the typing of the line. If this L command was omitted the first line containing READ would be printed repeatedly until you stopped the nonsense by typing ↑C ↑C to abort execution (then REENTER TECO-1Ø).

THE Q-REGISTERS

TECO uses another part of the computer core, in addition to the text buffer, for storage of text. This is called the Q-register buffer. There are 36 Q-registers each of which is given a single character name using the letters A-Z and the numbers Ø-9. In general the Q-register name will be referred to by the letter "i". A very useful feature of these Q-registers is that they can be used for storage of portions of text for later recall to the text buffer.

The Xi command (for te<u>X</u>t copy) copies text from the text buffer into Q-register i. In the form of its numeric arguments the Xi command is similar to the T command in that nXi copies all the text from the pointer through the n^{th} following end of line character into Q-register i and (m,n)Xi copies from the m+1st through the n^{th} character into Q-register i. It follows that HXi copies the whole buffer into Q-register i; HXi is equivalent to (Ø,Z)Xi.

The buffer pointer is not moved by the Xi command and the text in the text buffer is not altered in any way. However, the previous contents of Q-register i, if any, is deleted before the next text is copied. The only limit to the amount of text that can be stored in Q-registers is the amount of computer core space.

The notation Xi really stands for 72 different commands. The letter i is replaceable with 36 alphanumerics (alphabetic case is not significant) and each of these commands can be written in either the n lines form or the (m,n) characters form. The letter i is *not* an argument of the letter X; it does not have to be terminated with an ESCAPE.

The Gi command (for Get copy) copies the contents of the Q-register i back into the text buffer. There are no arguments associated with this command. The command Gi simply inserts a copy of the text in Q-register i immediately to the left of the current pointer position. After the insertion the pointer is located immediately after the end of the inserted text. The contents of Q-register i are not altered.

The Xi and Gi commands provide by far the easiest way of re-arranging parts of a file. For example, you may have a file with three sections and you wish to interchange the first and third parts. One way to do this would be to copy section one into Q-register A and delete section one in the text buffer. The pointer could then be moved to the end of section two and section one could be inserted here with the command GA. Section three could then be copied into a Q-register (you could use Q-register A again) and this section could be deleted from the text buffer. The pointer could then be moved to the start of the buffer and section three inserted here from the Q-register used to store it.

The Q-registers are a very powerful feature in TECO and other aspects of their use will be described in subsequent sections.

SAVING COMPLETED EDITING

It is a safe practice, especially when learning TECO, to copy, from time to time, the whole of the text buffer into a Q-register with the command HXi. If this is done regularly you will always have a copy of the text, with all but the last few changes, available for copying back into the text buffer in case of a disaster such as an unintended HK command. You need only use one Q-register for this as all you need is an updated edited version.

This procedure requires that you have adequate computer core available for your job but this should be the case since, as will be seen in the section on TECO input and output, there is no need to keep large amounts of text in the buffer. In fact, you should not do so.

DIRECT ENTRY TO Q-REGISTERS

In both TECO-1Ø and TECO-11 text can be copied from the command buffer to the Q-registers. The primary use of this feature is in the storage of commands for repeated use (described later) but it can also be used to transfer text to the Q-registers without going through the procedure of inserting it into the text buffer, copying it to a Q-register and finally deleting it from the text buffer.

The characters entered in the command buffer are not classified into commands and text; this only occurs during execution, which is started by double ESCAPE. Thus, text to be stored in a Q-register can be entered directly into the command buffer (after the prompt asterisk has been received). Now, if the "command string" is aborted by typing ↑G ↑G the characters which had been entered are not deleted (this was mentioned previously) but are retained until overwritten by subsequent entries into the command buffer. If the *first* command given after the asterisk prompt character which follows the ↑G ↑G is *i (copy from asterisk) the retained characters are copied to Q-register i. Note that the asterisk prompt will not serve as the first character of the command; you have to enter another.

In TECO-1Ø *i can be followed by other TECO commands in a command string, the execution of which is signalled by typing ($)($). In TECO-11 it is executed immediately (like the ? command after an error). This means that you must be careful to specify the correct Q-register the first time; there is no chance to edit this command. If you make a mistake you will overwrite the contents of the wrong Q-register.

TECO-11 provides a more conventional way of entering text into the Q-registers directly. This is through the ↑Ui (Update Q-register i) command. This must be entered in the CARET/U form because CONTROL-U will delete the current line in the command buffer. This command takes the form:

↑Ui . . . text . . . ($)

where . . . text . . . represents the string of characters which are copied into Q-register i (overwriting any previous contents). This form has advantages in programmed editing.

EXAMINING THE Q-REGISTERS

The basic way of examining the contents of a Q-register is to copy it into the text buffer and type it out on the terminal. This is the only way in TECO-1Ø. It is a little inconvenient if this text must then be deleted from the buffer and in this case the easiest approach is to copy the whole buffer into a spare Q-register and delete it before getting and typing-out the Q-register you want to examine. This can then be deleted and the original contents of the buffer can be retrieved. The pointer will be left at the end of the text.

In TECO-11 a much more convenient operation is defined. The command :Gi (modified Get Q-register i) types out the contents of Q-register i on the terminal *without* copying it into the text buffer or moving the pointer. The command :Qi returns the number of characters stored in Q-register i (but does not type it out). The command (↑Z) returns the total number of characters stored in all the Q-registers (it has a different function in TECO-1Ø).

SPECIAL TECO-11 BUFFERS

In TECO-11 the most recently entered search command text argument is stored in a special buffer and can be used in three ways. It can be inserted into the text buffer to the left of the pointer with the command G_ (Get UNDERSCORE) or, on some terminals, G← (Get BACKARROW). It can be typed out on the terminal without being entered into the buffer with the command :G_ (or :G←). Finally, it may be implied in search commands.

If a search command such as S, FS, (m,n)S etc. is entered without a text argument TECO assumes that the last search command text argument entered also applies to this command. For example, you may have given the command:

*JSDOGS ($)($)

and having found the first occurrence of "DOGS" have realised that you should have found the third occurrence. You could then give the command:

*2S ($)($)

and so achieve the desired result. This also works in TECO-1Ø but the stored string is not accessible.

TECO-11 also stores the negative of the length of the *most recent* insertion, text string *or* successful search test string. The insertion may have come from an I command, a G command, an FS command, an FN command (described in a later section) or an FR command. This information can be used in conjunction with the FR (Functional Replacement) command for correcting an unintended insertion. However you have to recognize that an error has been made before you make another insertion or search. You must also ensure that the pointer is positioned at the end of the incorrect insertion.

The actual number of characters involved in the last insertion can be ascertained with the (↑S) command which returns the negative of this number. The command

FR . . . text . . . ($)

is thus equivalent to

(↑S)D I . . . text . . . ($)

If it is desired to return the pointer to the start of the insertion which has just been made the command:

(↑S) C

can be given.

The command (↑Y) is equivalent to the argument:

(.+ (↑S) , .)

That is, it is the (m,n) argument which covers the part of the text involved in the last insertion or successful search, provided that the pointer is at the *end* of that string. If you have made an insertion in the wrong place and if you recognise the error before you make another insertion or successful search and if the pointer is at the end of the insertion (and if you remember the commands) you can give the command string:

(↑Y) Xi FR ($)

to store the wrongly inserted string in Q-register i and delete it from the text. On the whole it is preferable to remember to check what you're about to do before starting execution of a command string. Prevention is a lot better than cure.

In addition to these, TECO-11 also stores the most recent file specification. This is described later in connection with TECO input and output.

These features are not available in TECO-1Ø.

SECOND TECO EXERCISE

You should now repeat the things you have tried in the first exercise using additional commands. Insert some text (you could copy a few sentences from these notes) into the buffer and modify it.

The set of commands suggested previously was

J,ZJ,nC,nL,nT,HT,nD,nK,HK and I . . . text . . . ($).

These should now be supplemented with the commands

S . . . text . . . ($), FS . . . text 1 . . . ($) . . . text 2 . . . ($), Xi and Gi.

You could start off by making a copy of the text you insert into the buffer using the Xi command, if a disaster should occur a fresh copy can be obtained with the Gi command.

You should certainly try angle brackets and the semicolon modifier in this exercise. Numeric argument pairs and compound argument expressions may be tried as an optional extra.

With this set of commands you will be able to tackle quite complex editing problems. You'll also be able to make some monumental errors – this is part of learning.

The main deficiency at this point is that you haven't been given the commands which allow you to transfer text from the buffer to a stored disk file and vice versa. This is dealt with in the next section.

3. TECO INPUT AND OUTPUT

All the editing commands given so far make changes to the text in the buffer. The end purpose of this is always to create an output file in which the modified text is stored. Often the original text is entered from an input file in which it has been stored previously.

If you don't understand the input and output commands you will not be able to use TECO effectively.

TECO OUTPUT

You have by now learnt how to insert and modify text in the buffer. It is appropriate that you should now learn how to store the contents of the buffer so that they are not lost on exit from TECO. In general this involves three operations. First TECO must be told where the buffer contents are to be placed. Second, the buffer must be copied. Third, the completion of storage must be indicated. Note that the output operation is one of copying; the buffer contents remain unchanged.

The first of these steps is controlled by the EW (Enable Write) command. This has the form:

EWfilespec (ESC)

where the text argument, filespec, specifies the device on which output is to take place and the name to be given to the file created. The precise form of the filespec is system dependent and involves optional elements; the appropriate reference manual should be consulted for details. Every character between the EW and the ESCAPE is taken as the text argument (i.e. filespec). There can be no spaces between EW and filespec in TECO-11; they are allowed in TECO 1Ø.

All systems allow a form of filespec:

dev:filename.extension

in which dev: is the name of the actual device on which the operation will occur e.g. DX1: for the disk unit number 1 or TT: for the terminal printer. If the device is not file-structured (e.g. a printing unit) dev: is sufficient identification. However, with file-structured devices (e.g. disks), where a number of files may be stored together it is also necessary to specify the name given to the file as an identification (filename.extension). If such a file is to output on the current default device there is no need to include dev: in filespec.

Users of DECTAPE or magnetic tape should also consult the reference manual.

The EW command does *not* actually perform any output operations. Furthermore only one file at a time may be open for output. In TECO-1Ø an EW command automatically closes any output file previously enabled. In TECO-11 the open file must be explicitly closed or reference to it must be deleted (killed) before another EW is valid.

OUTPUT OPERATIONS

The fundamental output operation is specified by the command (m,n)P (for outPut). This is a copying operation (like (m,n)Xi) and does not affect the buffer or the pointer. When the command:

(m,n)P

is executed the $m+1^{st}$ through the n^{th} characters are copied to the currently enabled output file as specified by the latest EW command. If this file is left open subsequent output commands will append the specified output to the previously output text in this file.

The command HP copies the whole of the text buffer to the output file since H is equivalent to (Ø,Z).

As will be explained, it is desirable to separate the text in large files into "pages" with the FORM FEED character. This can be done by entering FORM FEEDS into the buffer. However, if it is desired to output the whole of the buffer with a FORM FEED appended this can be done directly with the special output command PW (outPut With FORM FEED).

Should the buffer be empty when the command PW is executed it will *not* copy a blank page to the output file. If you want to do this you will have to insert a FORM FEED in the buffer explicitly.

The command nPW will execute the PW command n times. That is, it will copy the buffer, with a FORM FEED, n times in the output file and leave the buffer unchanged. To repeat the HP command n times it is necessary to use the command form:

n<HP>

The command (m,n)PW is equivalent to (m,n)P. No FORM FEED is appended. This is for information only; it is preferable to reserve the PW command for the Put With function.

OUTPUT FILE CLOSURE

When output to a file is complete it is necessary to close it. This is particularly important with file-structured devices because a file which has not been closed is liable to be overwritten by subsequent operations on that storage unit. File closure can be specified explicitly with the command EF (End File). No filespec is required because it only operates on the currently open output file.

In TECO-1Ø every EW command automatically closes any currently open output file.

PROBLEMS WITH TECO OUTPUT

The EW command is interpreted in TECO as meaning that a *new* file with the specified filespec is to be written. You *cannot* open an existing file to receive more output. To achieve this you have to input the old file to the buffer (described later), output it to a new file, append the additions, close the file and finally, if necessary, rename with the old name outside TECO.

If an EW filespec refers to an existing file a warning message will be typed:

% SUPERSEDING EXISTING FILE

If this is unintended and not wanted you must take action immediately. This brings out the point that it is a good idea to give EW commands singly. If you give the command

EWfilespec $ (output commands) EF

the message will be given but it will be *too late*; the warning is not an error message and the file will be over-written with the EF command.

Recovery from this condition in TECO-1Ø is a little complex and requires commands which have not yet been discussed. In TECO-11 recovery can be made directly with the EK (Edit Kill) command; this leaves the file which is being superseded intact. A new output file can then be selected. It is important not to give an EF or EW command before this problem has been resolved. These commands will replace the old file with a new empty file of the same name. Recovery from this can be difficult.

The exception to this occurs with RSX-11. If the EW filespec includes a version number explicitly (by default the latest version is assumed) then any previously existing file with that specification is deleted from the directory at once. There is no recovery process short of examining the whole file storage device. The reference manual describes how TECO handles version numbers.

The TECO-11 EK command completely undoes the most recent EW command (provided that the file hasn't been closed). After an EK command the output file will not be listed in the storage device directory, even if output has already been made to it. An existing file of the same name will not be superseded. After an EK command that file can be reopened with another EW command. The EK command can be used to kill an *unclosed* file when, for example, the wrong output has been made to it. There is no provision in TECO for deleting closed files.

An output file should not be closed prematurely. If a file has been closed it is not possible to append to it directly. You will have to copy the additional material to another output file and merge it later using input commands. In addition, it is not possible to delete material from an output file selectively without inputting the file to the TECO text buffer.

The commands given to this point allow the creation and storage of new files. In TECO-1Ø this common situation is provided for directly with the monitor command:

.MAKE filespec <cr>

which is equivalent to:

.R TECO <cr>

*EWfilespec $

In TECO-11 under both RSTS/E and RSX-11 the equivalent system command is:

MAKE filespec <cr>

In TECO-11 under RT-11 the situation is a little different in that TECO is not the default editing program but in general is specified by the switch

/TECO

entered after an EDIT command (e.g. EDIT/TECO). TECO can be made the default editing program with the monitor command:

. SET EDITOR TECO <cr>

and when this is the case the /TECO switch can be omitted. Assuming that this has been done, the equivalent to the MAKE commands discussed above is:

. EDIT/CREATE filespec <cr>

TECO INPUT

So far the only way described of entering text into the buffer is through the terminal. This has been sufficient for the creation and subsequent storage of new files. However, the major use of TECO is in the editing of existing files. To do this the file which is to be edited must be copied into the text buffer; this is the only place where the text can be altered. This is done with the TECO input operations; the act of copying a file into the buffer is usually called reading. The input file is in no way affected by any of the TECO input operations.

Before a file can be read into the text buffer it must be specified so that TECO knows where to look for it. The ER (for Enable Read) command does this. It has the form:

ERfilespec $

where filespec follows the rules described in connection with the EW command.

The ER command doesn't actually perform input; it enables TECO to perform input from the specified file with subsequent commands. It does, however, terminate input from any previously specified input file. Thus only one file can be used for input at a given time. An ER command referring to the current input file re-sets it at the start.

PAGES

It is not necessary to copy the whole of a file into the text buffer at one time. The buffer is an area of the computer's core (memory) and it is wasteful to use too much of it. Besides, the input file will frequently be too big to fit in its entirety. Essentially there is no problem in editing large files with a small buffer; you read in part of the file, edit it in the buffer, output the corrected text to a new file, read in the next part and so proceed until whole file has been dealt with.

This is how TECO works and if you plan your editing carefully it works well. The problem in practice is that reading a file is not like leafing through a book; you can't go back. TECO input is a one-way process. When text is copied into the buffer from the input file you may imagine that a pointer is moved through the file keeping track of how much of it has been read. This pointer can only move forward through the file. If you have entered and edited part of the input file and then, for example, have unintentionally deleted the whole contents of the text buffer (and you haven't saved it in a Q-register) you can't immediately re-read the part you've lost. You have to go through an error recovery process which involves repeating some of what you had done before the mistake was made.

Within this mode of operation TECO works most efficiently when the number of characters in the text buffer does not exceed about 4ØØØ, or alternatively, when the buffer contains a maximum of 5Ø-6Ø lines of text. This is equivalent to a conventional page of printer output. It is, in any case, convenient to divide files into pages (pages are separated by a FORM FEED character) to improve their readability when printed. In TECO page divisions in the input file are used to keep the amount read into the buffer with each input operation at around the optimum.

It is up to the user to reduce the text stored in the buffer from time to time with output commands. If you don't delete or output text you can certainly use up all the available core space. This is expensive and wasteful in itself and generally slows the editing process.

It is best if input files are divided into pages of 6Ø lines or less but TECO *will* handle files which are not divided into pages. This is explained under input operations. If desired TECO can be used to divide a file into pages. It may be worthwhile doing this as a preliminary to the real editing task.

INPUT OPERATIONS

The basic TECO input command is the A (Append) command. This copies text from the input file and appends it to the current buffer contents. If the input file is divided into pages the Append command reads one page.

When TECO reads a page everything down to, but *not* including, the FORM FEED which marks the end of the page is copied. The FORM FEED never enters the buffer however an internal variable (flag) is set to show that the last input was terminated at the end of a page.

Since the end-of-page FORM FEED does not enter the buffer pages can be merged there directly. The A command does not take a numeric argument; nA is a different command. To append (and merge) n pages the command:

n<A>

must be used. After such an operation attention should be given to the size of the text buffer.

If the input file is not divided into pages the copying process will proceed until the end of the file is reached or, after the core available to the bufer is two-thirds full, either a LINE FEED is encountered or the buffer is within 128 characters of capacity. After such an operation it is advisable to check the size of the buffer (Z=) and if it is too big break it up into pages by inserting FORM FEEDs every 6Ø lines or so. It could then be output and the new file used as the input file for more efficient editing.

The A command has no effect if the end of the input file has previously been encountered.

If you wish to read the next page into the buffer without appending it to the previous contents the buffer should be cleared with the command HK before the A command is entered. The command string HKA is, for many users, preferable to the Y (Yank) command which is exactly equivalent. With HKA you must actually specify that the buffer is to be deleted; with Y your efforts can be destroyed before you realize what you have done.

In TECO-11 the action of the Y command can be modified through one part of the nED (Enable Defaults) command. If n=-1 Y works as described; if n=Ø or n=1 then, if no output file has been opened, Y works as described but if an output file is open and there is text in the buffer the Y command will not be executed and an error message will result. This reinforces the warning that Y must be treated carefully. Perhaps you should use HKA.

It will become apparent that in general editing it is not necessary for the user to know whether or not input was terminated by a FORM FEED in the input file. However, it is possible to find out. The command (↑E) returns -1 if the last input ended at a FORM FEED and returns Ø if this was not the case.

Another flag is set during input commands to indicate whether or not the last input operation was terminated by encountering the end of the input file. Again, this is not normally required by the user but if it is needed the command (↑N) will return -1 if the end of file was reached and Ø if it was not.

INPUT/OUTPUT PROBLEMS

The main problem with TECO input operations is to remain alert to the state of the text buffer. Should the buffer be cleared or should the next page be appended? Is the buffer getting too big? Should it be divided into pages? This is your responsibility.

The problem of how to deal with an undesired EW command was left unresolved for TECO-1Ø users. The question was what to do when an EW command is set to supersede a wanted file. In TECO-11 the EK command undoes this. In TECO-1Ø the output file which is to be superseded can be saved by first executing an ER command on it.

It is quite acceptable to issue ER and EW commands with the same filespec and to read from this file and output to a file of the same filespec at the same time. Physically a completely new file is being created by the output operations; when the file is closed this new file will be renamed and reference to the file actually used for input will be deleted from the device directory.

Thus, in TECO-1Ø, the file in question can be saved by transferring it through the buffer (without editing) to the new output file of the same filespec. That is, it can be completely copied. This is best done with commands which have yet to be described, but the principle has been established. The disadvantage of this procedure is that it may well be quite costly, though surely you wouldn't forget the filespec of a large and important file.

An alternative procedure is to leave the TECO program altogether with (↑C) (this does not close the output file) and re-run TECO (do not REENTER). This saves you from the need to copy the file but loses anything you may have had in the buffers. Anything output to the file named in the EW command will be lost but provided the file has not been closed with EF the old file of that name will not be superseded.

This description of the ER and EW commands conceals a hazard for TECO-11 users. When an EW command has been given, but no ER command has been given, then, after output is complete and the file is closed, the name of the output filename is added to the device directory. This causes no problems but the procedure is different if an ER command has *preceded* the EW command. When an ER command is given TECO copies the device directory and stores it internally. When output to a file specified by an EW command is completed the file closure operation *re-writes* the device directory using the stored directory with the reference to the newly completed output file added to it. This directory is lost and all files are disabled when you exit from TECO-11.

Now, in an RT-11 system it is occasionally useful to change disks during an editing job. If the disk is changed with the current ER command having been made on *first* disk and output is then made to the *second* disk and the output file is closed TECO will overwrite the second disk's directory with the directory referring to the first disk. To avoid this give an ER on a different *device* (any filename) then give an ER on the device with new disk.*

An ER command can refer to a disk, for example, in a non-file-structured way, i.e. ERdev:Ⓢ. In this way the whole contents of the disk can be read into the buffer regardless of file boundaries. This can be useful in recovering lost (ASCII) files. This is best done with commands which are yet to be described(TECO-11).

EXAMPLES OF FILE MANIPULATION

Creating a new file has been described. In editing an existing file the basic procedure is to read the old file (e.g. FILE.OLD) into the buffer a page at a time, make the changes desired, output the altered page to a new file (e.g. FILE.NEW) and repeat until all pages have been altered (if necessary) and output. FILE.NEW is then closed.

This procedure is recommended in preference to writing the output file with the same filespec as the input file (which you can do) since both the altered and the original files are kept. This is wise; you may have made an editing error which at worst may have produced an empty file. Don't delete the old file until you have verified that the new one is what you want. It is customary to rename old files with the extension .BAK to signify that they are back-up files. Renaming and deleting files is best done outside TECO.

It is a good idea to keep a master copy of important files and to perform editing only on copies of it. This may seem ultra-cautious but TECO does give you the power to wreak havoc on files.

Input and output does not have to be confined to one input file for one output file. Files can be merged quite simply. The single output file is specified in an EW command, the first input file is enabled with an ER command and the required parts of it are transferred through the buffer to the output file. The next input file is then enabled (the output file is left open) and it too is transferred to the output file and so on. Merging does not have to proceed page by page. The join can be made at any point by making use of the text buffer.

Splitting a file is accomplished by enabling the input file (the one to be split) with an ER command. The first output file is then specified with an EW command and the first part of the input file transferred to it. The second output file is the opened (in TECO-1Ø the second EW closes the first output file automatically), the next part of the input file transferred and so on.

If, in splitting a file, it is desired that there should be some duplication (overlap) in the new output files it is in general necessary to re-read the input file after the first output file has been closed and discard the unwanted part before creation of the subsequent and overlapping files can proceed. In particular cases the text and Q-register buffers may be used to store the overlapping part and so avoid the need to re-read the input file.

The commands described in this chapter form the basis of TECO input and output. You should practice using these before going on to the more complex commands.

THIRD TECO EXERCISE

The essential commands for TECO input and output are ER and EW for file specification, A to append a page from the output file to the buffer contents (or as much of the file as possible if it is not divided into pages), HP to output the whole of the buffer (leaving the buffer and pointer unchanged) and PW to output the buffer with a FORM FEED appended (if there is text in the buffer). The general command (m,n)P is usually used in a special form such as (Ø,.)P or (.,Z)P. You will also need EF and EK (TECO-11 only).

*Alternatively, leave TECO and REENTER (but this disables all enabled files).

The set of basic editing commands has now grown to consist of the following:

J, ZJ, nC, nL, nT, HT, nD, nK, HK, I . . . text . . . Ⓢ,
nS . . . text . . . Ⓢ, nFS . . . text 1 . . . Ⓢ . . . text 2 . . . Ⓢ,
Xi, Gi,
EWfilespec Ⓢ, (m,n)P, HP, PW,
EF, EK (TECO-11),
ERfilespec Ⓢ, A.

Use these commands to create a file (e.g. FILE1). You need only insert enough text into the buffer to allow the file to be identified by its contents, e.g. THIS IS FILE 1. Clear the buffer and create a couple more files. Now clear the buffer and read in FILE1; check it with an HT command. Append the other files to the text and output the merged files as a new file; clear the buffer and read and type out the merged file. Since it is not divided into pages this can be done with one A command.

Now make another new file by inputting FILE1 and outputting it with the PW command, clearing the buffer and repeating with the other original files. This new file will be a merged file with pages. Try inputting this file, checking the buffer after each A command to see how page by page input works. You could then split this file into others.

You will soon gain familiarity with these commands and in the process will learn the need to remain aware of the contents of the buffer during input and output operations. It is preferable to use the default devices for this exercise; if you don't you will almost certainly get the dev: part of the filespec wrong at some stage (usually forgetting to include it). This is the first thing to check when TECO can't execute an ER command on a file you've just created.

You should also become familiar with the TECO messages by giving an EW command while an output file is already open and also naming an existing (but unimportant) file. This will give TECO-11 users an opportunity to try the EK command. You should also make a simple multi-page file, as described, open it with an ER command and read one or two pages. Now give another ER command on the same (current input) file, clear the buffer, append a page and confirm that the file has been re-enabled at the start.

4. ADDITIONAL INPUT AND OUTPUT COMMANDS

The basic input/output commands enable you to perform all the essential file handling tasks in TECO. However, there are a number of very common editing tasks which require the execution of a string of these commands. TECO provides additional compound commands which can be used with greater convenience in place of these strings of basic commands.

THE NEXT-PAGE INPUT/OUTPUT COMMAND

Very often, when you finish editing a page, you will want to write the text in the buffer into the output file, appending a FORM FEED if the input was so terminated, clear the buffer and read in the next page from the input file. A single command does all this.

This command is nP. It is clearly quite different from the (m,n)P command and it is unfortunate that the same mnemonic form should have been used for it. It is recommended that nP should be thought of as standing for next Page.

If n = 1 or is omitted the next Page command will output and delete the current buffer (appending a FORM FEED if the last input operation was terminated by a FORM FEED) and finally will perform an append input operation (read in the next page). If n > 1 this sequence will be performed n times.

Like the PW command, nP will not output a completely blank page, although it can read one which is in the input file. Thus, nP will output and delete the current buffer contents, read in a new page, output that page (preserving the page layout in the input file) clear the buffer, read in a new page and so on. If a file containing many pages is being edited, you can skip over pages which don't require editing with the nP command. The output file written this way will be exactly the same as the input file except that any totally blank pages (i.e. consecutive FORM FEEDs in the input file) will not appear in the output file.

EXITING FROM TECO

The common way out of TECO is with ↑C, either as a command or as a character with immediate control function. In some systems ↑C allows current input/output operations to be completed before it has its terminating effect; this is not generally true of double ↑C. In TECO-1Ø ↑Z allows the completion of input/output operations but also has an effect (to be explained) on file naming. It is necessary to consult the reference manuals on these points.

The ↑C command is an abort command; in normal use you should allow TECO to indicate with the prompting asterisk that it has completed input/output before you enter ↑C. If you must abort input/output it will be your responsibility to check on how much actually took place. You should beware of the risk of missing or duplicated sections in files created during interrupted output operations. Files will not necessarily be closed after execution is aborted.

It may be noted that in TECO-1Ø and in TECO-11 under RT-11 the monitor command:

.CLOSE

can be used to close a file which was left open when the TECO task was terminated (e.g. by ↑C). However, there is no certainty that the output to the open file is complete in such circumstances; it is better to avoid mistakes in the first place.

The EX (for Exit) command provides a convenient way of terminating a simple editing job. It is another multi-function command whose exact operation must be understood. The EX command performs sufficient nP commands to transfer, firstly the contents of the text buffer, and subsequently the rest of the input file, to the output file (appending FORM FEED characters as indicated by the appropriate flag), then gives an EF and finally returns to the monitor with ↑C. The buffer is left empty.

TECO-11 provides an alternative form for use in systems where re-entry from the system is not possible (in TECO-1Ø and with RT-11 you can .REENTER the TECO program). This is the EC (Exit to Command level) command which is the same as EX except that TECO returns the asterisk prompt instead of exiting.

For example, to use TECO to make a copy of a file you could (after clearing the buffer) give the command:

*ERFILE.OLD (S) EWFILE.NEW (S) EC (S)(S)

or in some systems:

*ERFILE.OLD (S) EWFILE.NEW (S) EX (S)(S)

. REENTER

The nP command contained within the EX (EC) command will not output the initially empty buffer.

This is the safe way to finish a simple task of file creation or text correcting (on a single file). It ensures that an output command is given and results in an error message if no output file has been enabled.

It is not necessarily an appropriate command if you wish to merge, split or generally re-arrange several files all in the one TECO job.

EDITING WITH A BACK-UP FILE

The EW and ER commands are quite sufficient for editing a file. You could use an ER command and read (e.g. FILE.OLD) and after an EW command write into the corrected version as FILE.NEW. The uncorrected file, FILE.OLD still exists. The procedure usually adopted after such an operation is to rename the old file with the extension .BAK (for Back-up) and then rename the corrected file with the original name of the uncorrected file. That is, you start off with an incorrect file named (e.g.) FILE.EXT and end up with a copy of this incorrect file called FILE.BAK and a corrected file called FILE.EXT.

This is simply a safety procedure in case your editing was not wholly satisfactory in which case it may be simpler to re-edit the .BAK file rather than to try to patch up the faulty editing incorporated on the altered file.

In the RSX-11 system the procedure would be a little different since the filespec includes a version number. If this is not specified the latest version is assumed. Old versions are kept until explicitly deleted.

The EB (for Enable Back-up) command provides an easy way of performing the back-up file creation operation described above. The form of the command is

EBfilespec (S)

where the argument filespec is the file specification of the file to be edited. In the example above filespec would be FILE.EXT.

The EB command incorporates an ER command to enable reading of the input file and an EW command to hold the output. It also specifies special action when the file is closed. When a normal closure is executed (EF, EX, EC or EG – to be described), except in RSX-11, the output file is closed with the name of the original file specified in the EB command; this file is renamed with the extension .BAK. Any existing file with the same name and extension .BAK is deleted; only the most recent .BAK file of that name is kept.

In TECO-1Ø the command (↑Z) closes an output file but does not perform renaming. The output is contained in a file named nnnTEC.TMP where nnn is the user's job number. This temporary file can be read by an ER command but cannot be specified in an EB command. It will be overwritten by the next EB operation. Also, in TECO-1Ø the filespec in an EB command cannot refer to a file with the extension .BAK. If it should be necessary to read the back-up file (e.g. the editing performed on it was not satisfactory) you will have to use an ER command to read it.

In RSX-11 the EB command creates an edited file with a version number one higher than that specified in the filespec. Any number of previous versions may be kept within the limits of storage space.

To re-read an earlier (back-up) version in RSX-11 the specific version number must be given. The TECO-11 filespec storage buffer is useful in this situation. This stores the most recent filespec given in an EW, ER or EB command (also EI and EN, described later). In RSX-11 this filespec includes the version number even if it was not explicitly stated. It will be recalled that with an RSX-11 EW command the filespec is immediately entered in the file directory. An EW command is implied in the EB command and therefore the filespec in the filespec buffer after an RSX-11 EB command is the output filespec which has a version number one greater than the input file version number. Thus the full filespec of the input file in RSX-11 can be found by examining the filespec buffer and subtracting one from the version number.

The command which types out the most recent filespec is:

:G*

and this has no effect on the buffers or pointer. The command:

G*

copies the contents of the filespec buffer into the text buffer to the left of the pointer.

In all TECO-11 systems an EK command given before closure while an EB command is current removes all reference to the output file and leaves the input file open. That is, EK converts an EB command to an ER command.

The EB command is very useful for the common simple editing task in which a few changes are to be made to update a file. The standard procedure is to specify the file in an EB command, make the alterations and exit with EX. This is so common that all systems provide a system command to facilitate it. Thus:

.TECO filespec (DECsystem-1Ø);

.EDIT/TECO filespec or .EDIT filespec (RT-11);

TECO filespec (RSTS/E and RSX-11)

are equivalent to the appropriate system command to enter TECO followed automatically by the TECO commands

*EBfilespec $ HKA $$

Regardless of this endorsement, the EB command is not without its pitfalls. It is easy to forget that the output file contains only what you specifically output to it; EB does not imply an automatic EX command. Furthermore problems arise when only the most recent previous version is kept as a .BAK file. RSX-11 brings its own problems.

As an example of what can go wrong consider this sequence in editing a single-page file.

*EBFILE.EXT $ A $$

* (some editing) HK $$ (unintended deletion of all text in buffer)

*EX $$
. REENTER <cr>
} or * EC $$

The file FILE.EXT is stored but is empty. If you, unthinkingly, give the following commands

*EBFILE.EXT $ A $$

* (nothing in buffer, can't think why, give up) EX $$

You will be left with an empty file called FILE.EXT and an equally empty file called FILE.BAK.

It is not possible to execute an EB command while another EB command is still current. In TECO-1Ø this is explicitly invalid. In TECO-11 this is not possible because of the general rule against opening an output file while another remains open. This removes a very dangerous possibility of back-up file loss.

You can give ER commands while an EB command is current and can return (except in RSX-11) to the original input file because the input filename is not changed until the output file is closed. In RSX-11 it is necessary to refer explicitly to the correct version number.

Details regarding the operation of this command are given in the reference manuals.

INPUT/OUTPUT SEARCHES

All search commands which have been given to this point operate only within the text in the buffer. TECO also possesses commands which allow searches to proceed to search the rest of the input file if a match is not found within the buffer. Such a search command is the

nN . . . text . . . (Ⓢ)

(iNput search) command.

This command is analogous to the nS search command but if the search through the buffer is not successful it automatically performs an nP command to output the buffer (with a FORM FEED if the last input was so terminated), clear the buffer and append the next page of input. This page is then searched. If the search is unsuccessful the sequence is repeated until either a match is found, when the pointer is left at the end of the string of text which has been matched, or else the whole of the rest of the file has been examined unsuccessfully. In this event the buffer is left empty and (unless the command is in an iteration) an error condition results. The search command text argument must lie wholly in a page; it shouldn't include a FORM FEED as this might put it on two pages.

This command is not valid if no output file is open. In TECO-11 it can be of the reverse, -nN, or bounded, (m,n)N, forms. This does not, of course, imply that it can search back through the input file; the text in the buffer marks the limit of how far back the search can proceed.

This command has its functional variant (analogous to FS)

nFN . . . text1 . . . (Ⓢ) . . . text2 . . . (Ⓢ)

in which the search argument . . . text1 . . . is replaced with . . . text2 . . . if it is found. Its operation is otherwise as described for the N command.

These commands should not be casually substituted for the S and FS commands. If you use the wrong search text argument and so come to the end of the file before you have finished editing you will have to re-read the output file to get back to the point where you gave the unintended command.

An important use for the N command is in recovery from an output error. This may have been due to premature output file closure or occasionally a system fault. The result may be that you have file with a large amount of correctly edited material on it but it is not complete. The procedure here is to open this file for reading with an ER command and to open another for output. Clear the buffer (or save it in a Q-register first). Now, it is assumed that you know the last bit of editing which was successfully output (consult your memory or the terminal record). You should now find a unique piece of text within the last section of output and specify this as the search text in an N command.

The result will be that all the file output prior to the page identified with this search command is passed to the new output file. Since the page in the buffer was also successfully output previously it can be output again with another P command.

The problem now is to return to the original file which was being edited as the input file and to pass over the part which had been correctly edited. This is achieved with the UNDERSCORE (or BACKARROW) search command which has the form

n_ . . . text . . . ⓢ or n← . . . text . . . ⓢ

and is similar to the N search except that it performs *no* output. If the text is not found in the buffer the buffer is cleared before the next page is read in; it performs the sequence HKA instead of an nP operation. This command also takes the reverse and bounded forms but it does not have a functional replacement variant form.

This command can be put to good use in the situation described above. The last page which was correctly edited and output is identified as described above and the identifying text used in an UNDERSCORE search. When this search terminates this last corrected page will be in the buffer and all previous pages will have been discarded. Since this page has been corrected it can also be deleted and the next page read in with the HKA sequence. Editing may then proceed.

The UNDERSCORE search may also be used to recover from an unintended unsuccessful N search. The slight additional problem here is that before the input file can be read again from the start it will be necessary to issue another ER command on the original input file to re-open it.

SAFE EDITING

There are two causes of errors in TECO; your own mistakes and system faults. You should judge, in the light of your own experience, which is the more likely. Your own errors may affect the buffer or input/output operations. System faults are more likely to affect the input/output operations although a system crash will, of course, lose the buffer contents. In any case you should consider taking some precautionary measures against errors. The small cost and inconvenience involved is nothing compared to that required to make good a wrecked file.

Dealing first with the buffer, don't make insertion text arguments too long. Ten to fifteen lines has been suggested. Don't let the buffer get too big. Apart from the expense the more there is in the buffer the more there is to get lost. Until you gain experience you may find it useful to copy the buffer into a Q-register from time to time in case you give a deletion command by mistake; this does use core space but the cost is small in relation to the protection it offers.

Regarding output, it is particularly sad if an almost complete output file cannot be closed and is overwritten before it can be salvaged. The basic safeguard is to close it frequently. If you are making a new file you should give an EC (or EX) command to output the buffer and close the file every so often. You can then re-open the file with an EB command and return to point you had reached before. Clear the buffer explicitly (just to be sure) then give the command (e.g.) 1ØØP where the numeric argument is surely big enough to transfer the whole input file; nP has no effect after the end of file has been reached. A more elegant method (but one which has not yet been fully explained) is to clear the buffer and then read the input file page by page until the end of the file is encountered using the command:

<P(-(1+ ⓣN));> or in TECO-11, <P ⓣ_ ⓣN;>

You can then proceed to enter more text into the buffer and then into the file.

In normal file modification under an EB command it is safe practice to give an EC (or EX) command periodically. This ensures that the text in the buffer is output and also passes the rest of the input file to the output file, which is then closed and renamed. The standard precautions concerning the EB command must be borne in mind, particularly the fact that only one .BAK file is kept. In RSX-11 systems the number of stored versions must be kept under control.

THE EXIT AND GO COMMAND

In TECO-1Ø the command:

EG$ (for Exit and Go, *not* an enabling command)

first performs an EX command and then, instead of stopping with the system ready to accept commands, it automatically re-executes the last monitor command involving program compilation which was given before TECO was entered (using the same arguments, e.g. filespec, as was used before). The compile class commands are COMPILE, EXECUTE, LOAD and DEBUG.

The usual way of using this command is as follows. A program has been executed and then it is desired to change a parameter which cannot be entered into the program during its execution. The file (program) is immediately edited in TECO using an EB command and after the change has been made the command EG is given to exit and immediately execute the new version. This is a rather small convenience in normal editing but can be used in programmed TECO. This procedure can also be used when compilation has revealed errors. The error listing can be examined before the program is edited without affecting the EG command since this does not involve a compile class command.

In TECO-11 using RT-11 the command:

EG . . . string . . . $

performs an EX command and passes the . . . string . . . to the system as a system command. This may be a command in itself or the specification of an indirect command file. This is powerful in programmed TECO.

In TECO-11 using RSTS/E the command:

EG . . . string . . . $

performs an EX command and then checks the extension part of the most recently closed output file. If the extension is:

.CTL	the CCL command:	SUBMIT . . . string . . .	is executed;
.FOR	” ” ”	FORTRAN. . . string . . .	” ” ;
.MAC	” ” ”	MACRO . . . string . . .	” ” ;
.RNO	” ” ”	RNO . . . string . . .	” ” .

If the extension is not one of these four only the EX is performed.

In RSX-11 the EG$ command is identical to the EX command.

In TECO-1Ø and in TECO-11 under RSX-11 the EG$ command may be written without the concluding ESCAPE. In interactive editing it will be followed by two ESCAPEs since it will be the last TECO command executed but in programmed editing this is not so. In the interests of consistency the ESCAPE should be added so that in all systems EG may be regarded as a command which takes a text argument. In TECO-1Ø and TECO-11 under RSX-11 the argument is null.

SECONDARY INPUT AND OUTPUT STREAMS

TECO-11 under RSTS/E and RSX-11 allows for the specification of two input and two output files at the same time although only one of each of these can be *active* at one time. When TECO is entered the “primary stream” is active; output commands apply to the primary output file and input commands apply to the primary input file. To switch from the primary (default) output stream to the secondary output stream the command:

EA (for Enable Alternative output)

is executed. This doesn't open or close any files, nor does it affect the buffer; it simply means that future output commands such as EW, HP and EF will apply to the secondary output file. To change back from the secondary to the primary output stream the command:

EW $

is executed. Again, this doesn't open or close any file.

In this way you can open two output files and switch output between the two as desired. This can save a certain amount of closing and re-reading files during file splitting.

Analogous commands are applicable to input. The command:

EP (for Enable alternative inPut)

selects the secondary input stream. You can then open a secondary file with an ER command. The switch back to the primary input stream is achieved with the command:

ER (ESC)

which again has no effect on the files or buffer.

TECO only simulates two channels and these commands are only useful with disk files. The file handling procedures used in RSX-11 must be kept in mind. If a file has been opened for reading with an ER command and then an EW command has been given with the same filespec (including version number), the directory reference of the open input file is immediately deleted. If this is done input should not be switched away from the current input stream because it will then be impossible to open this file again (it has been deleted and superseded by the new output file with the same filespec).

"WILD CARD" FILE SPECIFICATION

TECO-11 under RSTS/E and RSX-11 allows incompletely specified files to be referenced. The details are system-dependent and the Reference Manual should be consulted. As an illustrative example, under the control of the EN command (described here) TECO interprets the filespec *.TEC as referring to the next file in the directory with any file name at all as long as it has the extension .TEC.

The only command which can accept such a "wild card" filespec is the command:

ENfilespec (ESC) (for Enable Next)

This is a preset or enabling command only and is used prior to the command:

EN (ESC)

which loads the filespec buffer with the next filespec which is compatible with the "wild card" filespec. Note that a full filespec is stored in the filespec buffer; this can be typed out with the :G* command.

If you are involved in a normal editing task and you want to examine or process all files which match a particular "wild card" notation it is probably easiest to use a series of EN (ESC) : G* commands and note the files you want to specify in later ER commands. You could do this directly using commands which have not yet been described but this is more appropriate in programmed TECO.

FILESPEC SWITCHES

All systems using TECO except RT-11 allow certain switches in file specifications. The general form is:

filespec/switch

The Reference Manuals should be consulted on these points but of particular interest are the switches

/ (that is, a blank switch)

in RSTS/E to allow easy editing of a BASIC-PLUS program file which contains continuation lines and the switches

/SUPLN and /GENLSN

in TECO-10 which respectively suppress and generate Line Sequence Numbers. The Line Sequence Number is a five digit number with leading zeroes followed by a TAB character. These are included in some files produced by other programs and can be suppressed on input by the first of these switches. The /GENLSN switch is used in conjunction with the EW (or EB) command to add them to file in the buffer which does not have them.

FOURTH TECO EXERCISE

The essential commands described in this chapter are nP, EB and EC (or EX then .REENTER). The N and _ or ← searches are all but essential; you will probably not feel the need for the others for quite some time.

The basic set of TECO commands is now complete:

J, ZJ nC, nL, nT, HT, nD, nK, HK, I . . . text . . . Ⓢ,

nS . . . text . . . Ⓢ, nFS . . . text1 . . . Ⓢ . . . text2 . . . Ⓢ,

Xi, Gi,

EWfilespec Ⓢ, (m,n)P, HP, PW, EF, EK (TECO-11),

ERfilespec Ⓢ,

EBfilespec Ⓢ, nP, EC (TECO-11), EX,

N . . . text . . . Ⓢ, _ . . . text . . . Ⓢ.

When you master this set of commands you will be able to tackle effectively almost any editing task you are likely to face. You will find as you read on that there are more powerful commands and that frequently used command strings can be stored ready for immediate use but these are really convenient extras; you now have been given enough information to be a TECO editor.

The additional essential commands described here must be practised. For a start the nP command should be tried. Make a simple three page file as described in the last exercise, clear the buffer and append the first page and then output it with the command P. Examine the buffer and proceed with P commands until the input file has been copied into the output file. Now check the contents of the output file and compare with the input file. You could repeat this using an A command in place of a P command to merge two pages.

The same input file can be used to try the EC (or EX) command. Pre-set the input file and open an output file, clear the buffer and give the EC command; examine the output file. For a variation, start with the first page already in the buffer before giving the EC command.

It is very important to understand the EB command. Make a simple file (e.g. "THIS IS FILE 1", named FILE1) and specify it in an EB command. Clear the buffer and read it in. Now insert the text "first modification" (for example) and output the buffer and close the output file either explicitly or with EC. Now examine the contents of FILE1 and also of FILE1.BAK. Users of RSX-11 should examine the latest two versions.

For a demonstration of the loss of a file give an EB on FILE1, clear the buffer, input the file and clear the buffer again. Now give the EC command; this simulates an unintended command sequence. Now give another EB on FILE1, clear the buffer and input and examine the file. Think about it. Give another EC command (this simulates lack of thought!) and examine FILE1 and FILE1.BAK. Where is the original FILE1? This does not apply in RSX-11.

The N and UNDERSCORE searches should be tried by making a simple multi-page file (PAGE 1, PAGE 2, PAGE 3 etc.) and following through the steps detailed in this chapter for returning to the previous editing point.

The EB, ER, EW and EK commands should be tried with an EB command still current to gain familiarity with the TECO messages.

PART TWO

ADVANCED INTERACTIVE TECO EDITING

5. Command Storage

6. Text Arguments

The information given in this part extends to cover most problems in interactive TECO editing. Some readers, for example those with terminals which handle all ASCII characters, will have little need of some of the features of TECO described here. Nevertheless it is advisable to read all sections to learn what can be achieved with TECO. On the other hand, the sections concerned with command storage and the literal specification of control characters contain information which is essential in advanced TECO editing.

The best way to learn to use these commands is to incorporate them into your editing procedures whenever you think they will be helpful.

5. COMMAND STORAGE

The Q-registers have, in addition to the features already described, the capacity to store commands. This facility is very important in programmed TECO editing. However, it is also extremely useful at this stage. An incidental but convenient use is in recovery from a common insertion error.

EXECUTION OF STORED COMMANDS

Command strings are simply strings of ASCII characters. They are entered into the command buffer and interpreted as commands during execution. However, the Q-registers do not distinguish character strings by their origin or purpose. Command strings can be stored in the Q-registers just as text can be stored. Furthermore the command Gi which copies the text in Q-register i into the buffer has an analogue in the command Mi.

This command, which stands for Macro (a word used to describe TECO command strings) copies the contents of Q-register i into the command execution string in place of the Mi command so that the next command executed when Mi is encountered is the first command in the string stored in Q-register i.

For example, the text buffer contains

ABC<cr><lf>DEF<cr><lf>GHI<cr><lf>

and Q-register D contains LTJL. The command

* JL2C MD ØLT (\$)(\$) is equivalent to

* JL2C LTJL ØLT (\$)(\$) and so will produce the following type-out.

GHI

DEF

*

The contents of Q-register i are not affected by the Mi command which is essentially a copying command. However the Mi command differs from the Gi command in that the text which was inserted in the command execution string is not entered permanently in the command string

buffer. To do this the limits of the temporarily inserted commands (text) must be recorded internally by TECO. This has two important effects. Firstly, the inserted command string must be complete in itself; TECO cannot look past these internal limits to interpret the command. This should be kept in mind when examining the rules for stored commands which are given below.

The second effect relates to the nesting of macros. Just as the command string in the buffer can "call" a stored macro with the command Mi, the stored macro can itself call stored macros with Mi commands. The restrictions are that each macro must be complete in itself and that because of the limitations of TECO's internal recording of where the temporary storage takes place, such nesting is limited to a depth of approximately ten levels.

When the Mi command is given the contents of Q-register i are copied. The contents of Q-register i can be changed in any way by the commands invoked by Mi because it is the copied text which is being executed and not the text actually stored in Q-register i. If changes are made to the Q-register referred to in the Mi command the original contents will be permanently lost since the copy which was placed in the command buffer is deleted after it has been executed.

STORAGE OF COMMANDS

The basic way of creating a stored macro is to enter it (as text) into the buffer and to copy it into Q-register i with an Xi command. This is not very convenient. There are occasions in programmed editing with TECO–1Ø when this procedure must be followed. In TECO–11 the ↑Ui...text...(\$) can be used instead.

However, in interactive TECO editing, as described so far, the easiest way of storing macros is to use the *i command described in a previous section. It has already been explained that if a command string is aborted before execution, by typing (↑G)(↑G), the aborted string can be transferred to Q-register i if the very next command given is *i. However, this command also applies in another case.

The second way applies after the command to be stored has been completed with a (\$)(\$) and execution has been initiated. If *i is given as the very *next* command the *whole* of the previous command string (less the second of the concluding ESCAPEs, which is not part of the string but simply signals that the string is complete for execution) is stored in Q-register i. This applies *even* if the command string contains invalid commands which caused execution to stop before the whole string had been executed.

In this way invalid command macros can be stored and entered into the text buffer with the Gi command, where they can be edited to produce a correct version which can be returned to the Q-register for execution.

The *i command must be first given after either the execution or abortion of the command which is to be stored. Note that the * prompt character, printed to TECO to indicate readiness for a command, will *not* serve as the * of this command. You have to type another asterisk.

Finally, and very usefully, although *i must be the first command executed after the command which is to store was executed or aborted, the error explanation commands ? and / can precede it.

RULES FOR STORED COMMANDS

Stored commands may quite legally contain single or double ESCAPEs. In TECO–1Ø, if a macro containing double ESCAPEs is executed as part of a command string execution will stop when this combination is reached.

Thus if Q-register A contains IABC (\$)(\$) the command

* MA ØLT ($)($), which is equivalent to
*I ABC ($)($) ØLT ($)($) will terminate after the insertion of ABC with no type-out.

However, in TECO–11 double ESCAPE has no significance. It should not, therefore, be used as a method of stopping execution.

In TECO–11 execution may be stopped from within a macro by including (↑C) in it. When this character is encountered TECO–11 immediately returns to the command level and the asterisk prompt is typed. You should note that the (↑C) command is different function when it is entered in the actual command string; in that case it results in exit from TECO. In TECO–1Ø you can make a somewhat improper use of (↑C) to abort execution from a macro. The (↑C) character can be entered either in the CARET/C form or by methods not yet described. When it is encountered execution ceases because it is not a legal TECO–1Ø command. An error message is typed and TECO returns to the command level and types an asterisk prompt.

This is preferable to the use of (↑Z) in TECO–1Ø to terminate execution from within a macro because (↑Z) has an effect on output operations and results in exit from TECO. It is preferable to the use of double ESCAPE for reasons of compatibility between TECO–1Ø and TECO–11.

The main restriction on the execution of stored macros is that the macro must be complete in the Q-register. It is perfectly legal to *store* an incomplete macro; it is not correct to try to *execute* it in incomplete form. The following examples illustrate this.

If I is stored in Q-register B the command *MB ABC ($)($) will *not* insert ABC into the text because the I command in Q-register B lacks a text argument and so is incomplete. An error message will result. On the other hand, if ABC ($) is stored in Q-register C the command *IMC ($)($) will *not* insert ABC into the text; it will insert the two characters MC and no error message will be given.

In contrast, a numeric argument is not an essential part of the following command in the way that a text argument *must* follow certain commands. Thus, a numeric argument may precede an Mi command and an Mi command (if it is one which ends with a command which returns a numeric value) may be used as a numeric argument for the following command. If Q-register A contains the single character J, the command

*ZMA ($)($)

will move the pointer to the end of the buffer. The same effect will be produced by the command

*MAJ ($)($)

if Q-register A contains the single character Z.

As another example, any stored group of repeated commands must be completely enclosed within a pair left and right angle brackets in the Q-register if it is to be executed by an Mi command. That is, it is *not legal* to try to express the command J 3<IA ($) L > J 3T ($)($) by storing 3<IA ($) in Q-register A and giving the command JMAL>J3T ($)($)

RECOVERY FROM AN INSERTION COMMAND ERROR

Q-register commands and storage can save you a lot of trouble if, in a long insertion command (you may be creating a new file), you find that you have forgotten to include the I which is necessary to indicate the start of an insertion string. In this case your text argument will be interpreted as a string of commands. If you execute this non-insertion command by typing ($)($) various odd commands will be executed until finally execution will stop when an invalid command is encountered and an error message will be typed.

However, all is *not lost*. If, for example, the command **A ($)($) is given immediately as the *next* command the whole command (and hence the insertion string) can be saved in Q-register A (in this case). The same could have been achieved if the error had been recognised before ($)($) had been given. In this case the string could have been completed, aborted with (↑G)(↑G) and then stored with *i.

With the command string safely stored in Q-register A the recovery process can proceed. First it is necessary to restore the text buffer to its condition before the undesired command was given (if necessary). This is where a recently updated copy in a Q-register (not A in this case!) is useful. The text can then be stored in a Q-register with an HXi command and the buffer cleared with HK. The stored command can then be copied into the buffer with the GA command, edited, like text, to insert the missing I at the start and then be returned in corrected form to Q-register A with the command HXA. The buffer is then cleared, the original contents of the text buffer restored with Gi, the pointer correctly positioned, and finally the corrected command can be re-executed with the command *MA ($)($). If the stored command consists solely of text to be inserted it is easier to move the pointer to the insertion point and give the command GA ($)($).

The procedure of inserting the missing I at the start of the command must be followed. It is not legal simply to store the text argument part of the command in Q-register A and give the command IMA because one of the rules of storing commands is that text arguments can't be stored separately from their commands.

This is certainly extra work but it can be a lot easier than starting again from scratch. It does depend on your immediately recognizing that the insertion string was in error. You must take note of error messages!

You may at some stage find that an insertion string is correctly executed even though the initial I has been left out. This will occur if the first character in the string is CTRL-I (or <tab>), as is not uncommon in lines of FORTRAN programming.

This occurs because the command I (↑I) string ($) can be replaced by (↑I) string ($). This is a totally unnecessary command. It would have been a lot nicer if (↑I) could have been reserved for formatting command strings.

RUNNING OUT OF CORE SPACE

The ultimate amount of core space available to the TECO buffers (e.g. text, command and Q-registers) depends on the system as a whole. It is definitely a finite quantity. TECO automatically swaps core space between the various buffer areas as required, however, a command to insert more text into any of the buffers may have the result that all available core space is used.

The various versions of TECO give different responses to commands which would exceed core capacity. If a command is being executed when capacity is exceeded execution stops and an error message is typed (?COR STORAGE CAPACITY EXCEEDED in TECO–10 and ?MEM MEMORY OVERFLOW in TECO–11). Capacity can also be exceeded while typing a command (for instance a long insertion text). With TECO–11 the terminal bell rings when capacity is exceeded this way and you should delete a few characters to shorten the command. You should also examine the text buffer to see if some of it should be output before you continue. With TECO–10 you do not get this warning.

The core is not necessarily absolutely filled when these warnings are given. You will generally find that you have enough space to output some text or to reduce the amount stored in the Q-registers. You must take note of the message. Given sufficient determination you can, by ignoring them, totally use up all core space so that no TECO commands can be executed. If you do this you will need to use operations outside TECO to try to retrieve the situation.

FIFTH TECO EXERCISE

This is a short chapter but very important. You should take particular care to confirm the rules for using stored commands through practical tests. It is easiest to store the macro with the *i command; the problem at this stage with storing commands through either the buffer and the Xi command or with the ↑U command (TECO–11) is that you will not be able to include ESCAPE characters in the text. This is covered in the next chapter.

Enter the command IA($) in Q-register A and clear the buffer. Give the command MA and examine the buffer. Repeat this with the command 5<MA>. Now store the command <IA($)> in Q-register B and give the command 5MB. All these forms are acceptable because the macro is complete in each case. You might confirm that it is not correct to store IA($)> in Q-register C and then give the command 3 <MC. Neither is it correct to store A($) (Q-register D) and give the command IMD.

With the Q-registers left as described above store the command MA in Q-register E. Since no ESCAPE characters are involved you can do this by clearing the buffer, inserting MA, copying this to Q-register E and then clearing the buffer (or use ↑UMA($) in TECO–11). Now give the command 4<ME> and satisfy yourself that you can explain what happened. Repeat this command, use the command *F to store the command string in Q-register F, clear the buffer, copy the text in Q-register F into the buffer with GF and examine it to confirm that the command string is not permanently altered by the replacement of the Mi commands with the stored command strings during execution. TECO–11 users could use the command :GF.

You should also confirm that a TECO macro can modify the Q-register in which it was stored. To show this store the following command in Q-register G:

```
HK IOVERWRITTEN($) HXG HK
```

Now execute this macro and then examine the contents of Q-register G.

Finally, you should confirm that a macro can return a numeric argument. Store 3 in Q-register H and then give the command MH=. Of course, if the macro ends with an angle bracket it cannot return numeric values.

Don't try to run out core space unless you have money to burn.

6. TEXT ARGUMENTS

Insertion and search commands require text arguments. In the editing described to this point these have been simple and definite. However, definite arguments are unnecessarily restricting; it may be satisfactory to search for, say, just the first alphabetic character (whatever it may be) in a file consisting mainly of numeric data. The commands described in this section allow this.

Alphabetic case is another consideration in text arguments. Alphabetic case control is also covered here.

The remaining problem with text arguments is the literal specification of certain characters. For example, when extensive use is made of stored macros the need to edit them will soon arise. This is done by copying the macro into the text buffer and editing it there using normal TECO editing techniques. The problem with this is that macros often contain special characters which cannot be specified in text arguments using the commands described previously. The most common example is provided by the ESCAPE characters which occur in almost all macros. The problem here is to specify the ESCAPE character literally so that it can be entered normally without signifying the end of the argument.

THE AT-SIGN MODIFIER

The problem of including ESCAPE characters in text arguments is so common that a special procedure has been included in solely to deal with it. This is the @ (AT-sign for Alternative Text) modifier. This applies to insertion and search arguments in TECO–10 and to *all* text arguments in TECO–11.

The @ is entered immediately before the first letter of the command which it modifies (i.e. after any numeric argument), e.g. @I, @S, n@FS or @ER (TECO–11 only). The effect is that the *first* character after the command is taken to indicate what the delimiting character for the text argument will be. It is not taken as part of the argument. The text argument is then terminated by the *next* occurrence of this first character (again, it is not taken as part of the argument. In this form ESCAPE has no delimiting function and can be included in the text. The general form of such a command is, for example

@S /...text.../

where / is arbitrary but must not occur within ...text... . It is common to use the slash, /, as the delimiter. No ESCAPE character is required when this form of text argument is used.

The FS command also takes this form:

@FS/...text 1.../...text 2.../

and deletion without replacement can readily be specified:

@FS/...text...//.

Double ESCAPE cannot be entered directly but the practice of including this combination in macros is not recommended anyway.

THE ASCII INSERTION COMMAND

The @ modifier applies *only* to the inclusion of ESCAPES in text arguments. However there are other characters which cannot be included in text arguments in the usual way. These fall into three classes: those which have special TECO functions inside text arguments (described in this chapter); those which have immediate consequences (such as RUBOUT) and, to an extent, those which affect the terminal carriage mechanism. TECO deals with these in a number of ways.

Firstly, *any* ASCII character can be inserted into the text buffer and from there combined with text or with commands. This may not be very convenient but with some characters it is the only way. Once a character is in the text buffer it causes no problems; the problem is to get it into the command string in a literal form. The command which allows any ASCII character to be entered into the text buffer has the form

nI (\$)

where n is the decimal ASCII code of the character and the terminating ESCAPE character is mandatory. This inserts the *single* specified character into the buffer to the left of the pointer.

This command may be used to insert carriage control characters without causing any effect on the terminal. The characters which are the most likely candidates for this treatment are listed here:

Character	Code (decimal)	Character	Code (decimal)
NULL	Ø	BACKSPACE (↑H)	8
↑C	3	TAB (↑I)	9
↑G	7	LINE FEED (↑J)	10
↑O	15	VERTICAL TAB (↑K)	11
↑U	21	FORM FEED (↑L)	12
ESCAPE	27	CARRIAGE RETURN (↑M)	13
RUBOUT	127		

NULL is a special character which can be inserted and output but which is not read by input operations. Combinations such as ↑G ↑G need not be entered this way; they can be entered with other techniques provided an arbitrary character is entered, and immediately deleted, between them.

It has not yet been explained why there is a need for the literal specification of other characters. This will now be clarified.

INSIDE-TEXT-ARGUMENT COMMANDS

TECO possesses a class of commands which operate only when they are given inside a text argument. Some of these also function as quite different commands when entered outside text arguments. There is no conflict in this; TECO recognizes when characters in a command string are inside or outside a text argument.

If these commands are to be useful they must be capable of being edited. The problem is that if, for example, a certain character has a command function inside an insertion text argument and you wish to insert it into a macro using TECO insertion commands you have to specify it literally. Otherwise, since you are entering it in an insertion argument it will exercise its command function when you try to enter it.

In fact, in TECO–11 no such commands are defined for insertion text arguments (they are in search arguments). In TECO–1Ø a number are defined for the purpose of allowing texts which contain both upper and lower case characters to be edited from a terminal which possesses only one case (usually upper case). Although, in TECO–1Ø, only a few commands are *defined* as inside-text-argument commands a large number have been set aside for future developments and so are classed as inside-text-argument commands. All control characters except ↑C, ↑O, ↑U and the carriage controls ↑H through ↑M are so reserved.

This means that there is an important difference in this regard between TECO–1Ø and TECO–11. In TECO–11 the only characters which require literal specification in insertion

commands are those with special function discussed in the previous section. In TECO–1Ø all reserved characters must be specified literally when they are included in insertion text arguments if they are not to be interpreted as commands.

In search text arguments the situation is different. In TECO–1Ø the set of characters defined above are reserved as commands but in TECO–11 a (smaller) set of defined characters also have command functions. Thus in both versions there are a number of characters which may require literal specification in search text arguments.

It is certainly possible, and occasionally appropriate, to construct a search text with literal specification of inside-text-argument commands by inserting the search command into the buffer using the ASCII insertion command, transferring this to a Q-register and then executing it as a macro. As a general method this is far too inconvenient. Accordingly special literal specification commands have been defined. These apply *only* to defined or reserved inside-text-argument commands and do not operate on those special characters which can only be inserted with the ASCII insertion command.

Before describing these the operation of some inside-text-argument commands will be illustrated.

COMMANDS INSIDE SEARCH STRINGS

Sometimes when searching for a character string it may be convenient to accept any character as a match in a particular position in the string. For example you may wish to search a text for the first four letter word ending in AND. The search string to achieve this can be written using the inside text command (↑X) (read as character). This causes any character at this position in the string to be accepted as a match. However, there must be a character at that position. Thus, for the example described, the command would be:

S (↑X) AND($)

On the other hand, it may be desired to accept any character as a match at a particular position in the search string except for one particular character. The inside text command (↑N) *x* (read as Not *x*), where *x* is the character *excepted* from constituting a match, will achieve this. For example if it is desired to search a text for the first four letter word beginning with S and ending in ND but not including the word SAND, a suitable command would be:

SS (↑N) AND($)

The command (↑S) causes TECO to accept any separator character as a match at this position. A separator is any character except a letter or a digit; in TECO–1Ø the DOLLAR, PERCENT SIGN and PERIOD (FULL STOP) are not acceptable as separator characters with this command.

In some systems (↑S) has a special function which precludes its use within TECO. It may be possible to disable this function (e.g. in some systems, with the monitor command .SET TTY NO PAGE) after which this command will be usable in TECO.

TECO–11 provides an alternative procedure for entering such control characters into text arguments. This is the CARET (or UPARROW) command which works exactly as described previously for normal TECO commands in that, during execution, the character following the CARET is interpreted as a control character. This command is actually available as a selectable option controlled by the ED command (described previously). It is not available as the default option in all systems but applies if the ED value is -1 or Ø ; the command 1ED disables this option.

By using this form of command you can enter CARRIAGE RETURN as CARET-M to avoid having a carriage return action echoed while the command is being entered. TECO-1Ø

users would have to use the nI($)insertion command to achieve this.

The other special inside text match control command is the (↑E)*x* command which has a number of variants. In each variant form the effect is to accept as a match, at the specified position on the search string, the first occurrence of a character from one of the following groups specified in the command. The following forms are common to TECO–1Ø and TECO–11:

(↑E)A	any alphabetic character (A–Z and a–z).
(↑E)D	any digit (Ø–9).
(↑E)L	any end of line character (LINE FEED, FORM FEED, VERT TAB, end of buffer).
*(↑E)S	any *string* of SPACE and/or TAB characters.

Teco–1Ø allows a number of other forms of this command:

(↑E)V	any lower case alphabetic character (a–z).
(↑E)W	any upper case alphabetic character (A–Z).
(↑E)<nnn>	the ASCII character whose *octal* code is nnn.
(↑E)[a,b,c,..]	any one of the ASCII characters a,b,c, etc.

TECO–11 also has a set of these commands which are not available in TECO–1Ø:

(↑E)C	any RAD5Ø character (i.e. any alphanumeric, PERIOD or DOLLAR).
(↑E)R	any alphanumeric character.
(↑E)X	equivalent to (↑X).

Only TECO–1Ø provides the direct equivalent of the ASCII insertion command. This is the (↑E)<nnn> command which allows any ASCII character to be inserted into a search text argument literally. In TECO–11 the problem is usually resolved through the use of the CARET command inside the text argument. You can search for CONTROL/C in the text with the command CARET-C. The only difficulty arises with a search for RUBOUT which is not a CONTROL character. If it is not possible to search for a known context the search text argument must be constructed indirectly via the Q-registers as described above.

TECO–11 possesses three special commands which allow stored character strings to be inserted directly into search text arguments. The command:

(↑E)Qi is used to specify that the text stored in Q-register i is to be taken as part of the search text argument in the place occupied by this command. The command:

(↑E)Q* has the same effect but using the text stored in the filespec buffer. The command:

(↑E)Q_ (or (↑E)Q←) is similar but uses the text stored in the search string buffer thus the command string:

S(↑E)Q_($) is equivalent to S($).

It is important to realize that when these special search-text-building commands are executed the string stored in the search string buffer includes the text which has been inserted into the search argument. For example, assume that the last search command argument was XYZ. The command:

SABC(↑E)Q_ DEF($)

* If the last character in the buffer is a SPACE TECO-11 will not accept it as a match with this command.

will search for the string ABCXYZDEF. If the next search command is also:

SABC (↑E) Q_DEF($)

the string sought will be ABCABCSYZDEFDEF.

The TECO–11 restriction on the length of a search text string to a maximum of 128 characters to be *matched* still applies, even though with these commands it may take more than one character in the text argument to define one character to be matched. In TECO–1Ø, where the matching limit is 36 characters, the search text argument is limited to a maximum of 8Ø characters including those needed to define the characters which are to be matched.

The other inside-text-commands are associated with alphabetic case control or literal specification.

INSIDE-TEXT LITERAL SPECIFICATION

There is evidently a large number of characters which have special command functions when used inside search text arguments in both versions of TECO (and inside insertion text arguments in TECO–1Ø). If it is desired to search for, say, the character (↑X) it must be specified literally; if it isn't it will be taken to indicate that any character will match at this position.

Literal specification is indicated in both versions with the inside-text-argument command (↑Q), although, as explained below, it is preferable to use the equivalent form (↑R) in TECO–1Ø. The occurrence of (↑Q) (or (↑R) in TECO–1Ø) indicates that the following character is to be taken literally. Thus, to search for (↑X) the following command string can be used:

S(↑Q)(↑X)($).

In TECO–11 (↑Q) need only be used in search text arguments (there are no inside-insertion-text-argument commands in TECO–11) and can also be used to specify ESCAPE literally and provides an alternative to the use of the @ modified text argument form. In TECO–1Ø (↑Q) does not apply to ESCAPE but the command (↑R) does.

The choice of (↑Q) for this command, which can be fairly commonly used in editing macros, is a little odd since, like (↑S), it may be a system command and not automatically available in TECO. Depending on the system, this character may be freed for use in TECO with a system command (e.g. .SET TTY NO PAGE). If this is not possible there are alternative procedures.

In TECO–11 the (↑Q) command may be given in the alternative CARET-Q form. In TECO–1Ø the alternative command (↑R) is preferable in any case since it is directly equivalent to (↑Q) in TECO–11.

In a search command the sequence ↑Q (↑X) specifies that the single character (↑X) is to be matched. The sequence ↑Q↑X means that the single character ↑ (specified by ↑Q↑), lying before the single character X, and the character X are to be matched. The second form will fail if the text being searched contains only (↑X) and vice versa. The indiscriminate mixing of CARET and CONTROL forms can thus lead to problems in subsequent editing. The CARET form is not available as an inside-text-argument command in TECO–1Ø.

TECO–1Ø has literal specification command which applies to whole text arguments and not just to the next character. The command (↑T) in a text argument causes all succeeding characters with command functions in the text string to be interpreted literally. This interpretation is cancelled either by another (↑T) character or when the end of the argument is reached. This command does not apply to ESCAPE, (↑R) or (↑T) itself.

This means that it is only necessary to enter a single (↑T) character at the start of the text argument to ensure that all characters within it (except for special characters, (↑R), (↑T) and ESCAPE) are interpreted literally. The argument is terminated normally with an ESCAPE. If it is wished to include the characters (↑R), (↑T) or ESCAPE literally within the text argument they may be preceded by (↑R), which does not affect the (↑T) function which may be current and applies only to the single following character. However, like (↑Q) and (↑S), (↑T) may be a system command which will have to be disabled by the monitor command .SET TTY RTCOMPATIBILITY before it can be used in TECO.

ALPHABETIC CASE AND TERMINAL CONTROL

TECO commands may be given in either upper or lower case alphabetic characters but alphabetic case is an important consideration in text arguments.

Not all terminals are equipped to transmit or print lower case letters; if these letters are not available it is generally the case that all 31 ASCII characters with decimal codes 96 to 126 are missing, i.e. the last 32 characters in the set with the exception of RUBOUT. Even if the full ASCII set is available on the terminal it will be necessary to ensure that the system is set to receive lower case characters (e.g. by a monitor .SET TTY LC command).

Within TECO itself transmission to and from the terminal is controlled by the nET (Enable Terminal) command. In TECO–10 this command only affects the transmission of information to the printer. If n is equal to zero control characters are echoed in the CARET form as described, except for ESCAPE which is echoed as a DOLLAR; this is the default option. If n is non-zero characters are transmitted literally; non-printing characters are not printed. If the terminal is one which does print special symbols for control characters these will be printed.

The nET command has much more to it in TECO–11. In TECO–11 the ET value is stored as a bit-encoded word, i.e. it is stored as a string of Ø's and 1's but significance is attached to whether or not particular bits are zero or one rather than to the number which these Ø's and 1's represent in binary form. If all bits are set to zero TECO–11 commands which involve the terminal proceed as described. This is not necessarily the default setting.

If Bit Ø is set literal transmission occurs. If Bit 2 is set (i.e. if only this bit is set the value of ET is $2^2 = 4$) lower case characters are input as lower case. Normally in TECO–11 characters are inserted in upper case form regardless of how they are entered in the text argument. When this bit is set lower case characters can be inserted exactly as they are entered.

Bit 1 (ET = $2^1 = 2$) relates to V.D.U. terminals, Bit 4 (ET = $2^4 = 16$) controls the operation of the CONTROL-O character (described previously). Bit 8 ($2^8 = 256$) truncates output lines to the terminal's width (RSTS/E and RSX–11). Other bits relate to commands which have not yet been discussed. To set more than one bit of the ET word the argument should be entered as the sum of the numbers corresponding to the desired bits. For example, to set Bit 2 and Bit 4 simultaneously (and no others) use the command:

```
(4+16)ET
```

The command ET returns the current value; after the command shown above it would return 2Ø. To set Bit 2 and Bit 4 in addition to whatever bits are already set use the command:

```
(ET#(4+16))ET
```

The hachure (#, "hash") means bitwise logical OR; it is described in a later chapter. To switch a bit off it is essential first to ensure that it is actually set. For example:

```
(ET#16-16)ET
```

sets and switches off bit 4.

It will be necessary to consult the reference manual for full details of the default settings. These depend on the operating system and the terminal equipment.

To summarize, if you have a terminal which handles lower case letters and if the system is set to accept them and if, in TECO–11, Bit 2 of the ET word is set, then you will have no trouble in inserting text in either alphabetic case.

CASE FLAGGING DURING TYPE-OUT

Another consideration in TECO's handling of alphabetic case concerns type-out. TECO provides for the type-out of lower case characters on terminals which print only the upper case forms by "case flagging". The lower case characters are distinguished by printing an apostrophe immediately before the equivalent upper case character. Thus, the lower case character "a" is typed as " 'A ". This is called lower case flagging and is controlled by the nEU (Enable Upper/lower) command.

Lower case flagging is specified when the argument is equal to zero. This is the default setting in most (not all) systems. The current value is returned by the command EU with no argument. All 32 characters with decimal ASCII codes in the range 96–127 are considered to be lower case characters; the upper case equivalent is the character whose ASCII code number is 32 less. Thus, for example, left curly bracket is the lower case equivalent of left square bracket.

This option is inappropriate if the terminal has the ability to print lower case characters. It is automatically overridden if the ET command is set for literal transmission but this may not be a desirable option either. The appropriate option in this circumstance is for case flagging to be disabled. This is done by specifying n=–1 in the EU command.

If case flagging is disabled with an upper–case–only terminal lower case characters are printed as their upper case equivalents and are not distinguished from them.

Upper case flagging is selected by setting n=1. This is useful with an upper–case–only terminal when the text being typed is mainly in lower case characters; these are typed in upper case equivalent form and the smaller number of actual upper case characters are distinguished by being preceded by an apostrophe.

CASE CONTROL IN SEARCHES

If the system and TECO are equipped and enabled to handle both alphabetic cases specifying case in search command text arguments is straightforward. In both versions of TECO the default option for search commands is that case is disregarded in seeking a match, i.e. if the text argument contains the letter "A" either "A" or "a" in the text will be a satisfactory match. The same will be true if the search argument was "a". This is called "either–case mode". This option is controlled by the n(↑X) command. In this application the (↑X) is used outside the text argument and is quite different in its effect from the inside–text–argument command of the same form.

If the argument, n, of this command is zero either–case mode (the default option) is enabled. If n=1 "exact–case mode" applies to searches. In this mode a match is achieved only when the alphabetic case of the character in the text is the same as that used in the search argument. The command (↑X) with no argument returns the current value of the case mode setting.

In TECO–1Ø an additional inside–text–argument command is defined for use in conjunction with the case mode command, n(↑X). This allows the case mode setting to be varied within a search text argument.

If the exact-case search mode is currently operative it is possible to specify that some characters in the search string may be accepted in either-case mode by enclosing them with (↑\) commands. Thus the command

*1 (↑X) JSABC (↑\) DEF (↑\) ($)($)

will be successful if it locates a string in which ABC are in upper case and each of the following characters DEF is in either upper or lower case. For example, it will be successful if it locates ABCDEF or ABCdef or ABCDEf of ABCDeF, etc.

CASE CONTROL WITH LIMITED TERMINALS

In TECO–1Ø full provision has been made to allow editing of texts which contain characters whose ASCII codes are in the range 96–127 (decimal) from terminals on which these characters are missing. In practice the procedures are rather cumbersome and unless the need is pressing it is better to save the task of editing such files until access to a full ASCII terminal can be obtained.

In TECO–11 the task described above is more difficult. A full ASCII text can be searched satisfactorily but lower case characters can only be inserted from an upper case terminal with the ASCII insertion command nI($). The easiest way to do this is to memorize the ASCII codes for lower case characters. Another way is to use the (↑↑) x command which returns the code of the character x. Thus to enter lower case "a":

((↑↑)A+32)I($)

Since the pointer is left at the end of an insertion it is no problem to form a long insertion string by a sequence of such commands.

Inside TECO–11 search text arguments the (↑↑)x command has the specific function of specifying that the character to be matched at this position is the lower case equivalent of the character x. All 32 characters with decimal ASCII codes 96–127 can be specified in this (but only in search text arguments). This is the extent of TECO–11 provisions for editing full ASCII texts with limited terminals.

In TECO–1Ø the inside-text-argument command (↑↑)x functions as described above but is limited to the specification of *non*-alphabetic lower case characters. It also differs from the TECO–11 command in that it can be used in both insertion and search arguments.

In TECO–1Ø case conversion of *alphabetic* characters is specified by a number of commands using two control characters. These are (↑V) which specifies lower case and (↑W) which specifies upper case. However there are three forms in which these commands can be used. Case control applies only to alphabetic characters and only within insertion and search text argument strings.

The default condition is for no conversion to take place. Characters are taken to be of the case in which they are entered. With the terminal equipment under discussion this is upper case. The command (↑V) *outside* a text argument means that all characters are to be taken to be in the lower case form, unless overruled by the higher priority commands which are given inside text arguments. This is quite sufficient if only insertion commands are to be given. However the default search setting is either-case mode so this command will only be significant in searches if used in conjunction with the 1 (↑X) exact mode command. The default translation mode can be restored with the command Ø(↑V).

There are analogous commands (↑W) and Ø(↑W) (also for use outside text arguments). These are not required with terminals which handle upper case only.

The (↑V) command *outside* a text argument applies until cancelled (by Ø (↑V), (↑W) or Ø (↑W)) but is rather weak. A double (↑V) command (i.e. (↑V)(↑V) occurring *inside* a text argument overrules any general (outside text) conversion command which is current but, unless overruled for a single character, specifies that all succeeding characters *within that string* only are to be interpreted as lower case. The (↑W)(↑W) command *inside* a text argument has an analogous effect in causing characters to be taken to be upper case. This is useful, even with an upper-case-only terminal because it overrides any general (↑V) conversion command. For example, the command

*_ (↑V) IABC (↑W)(↑W) DEF ($)($)

inserts the string abcDEF. This could also be achieved by the command:

*_ I (↑V)(↑V) ABC (↑W)(↑W) DEF ($)($)

When conversion commands are given *inside a search* text argument then *exact-case* search mode applies to all characters under control of the inside text translation command. For example the search command

*_ (↑V) JSABC (↑W)(↑W) DEF ($)($)

will be successful if the text contains any of the strings abcDEF or ABCDEF or abCDEF etc., but not if it contains only ABCdef or AbcDeF.; the DEF string must be in upper case.

The highest priority conversion command is the single (↑V) or (↑W) inside a text argument. This converts the next character *only* to the specified case. It overrules any current double character conversion command inside the text and any general conversion command. Again, exact case search mode applies for the character whose case is so specified.

The command:

(↑V) 1 (↑X) S (↑W) A (↑\)(↑\) ASE (↑\) O (↑\) F (↑W)(↑W) TECO (↑\) C (↑V)(↑V) ONVERSION ($)

will be successful if the text contains:

"A case of TECO conversion" or "A Case Of TECO Conversion" (or variant forms of these permitted by the optional initial capitalization). Note that if the command 1 (↑X) had not been given all characters before "TECO" would have been matched in either case mode.

The commands (↑V) and (↑W) have different meanings in TECO–11. The command (↑W) applies only to RT-11 with the VTII graphics processor (see Reference Manual).

SIXTH TECO EXERCISE

A large number of commands have been described in this chapter but it isn't very likely that you will need all of them. The most important new concept introduced here is that there are a number of inside-text-argument commands which are in general quite different in effect from the commands represented by the same characters when used outside text arguments. In TECO–1Ø an extensive range of control characters has been reserved for this purpose in search and insertion text arguments and these can only be expressed in the CONTROL/character form. In TECO–11 there is a smaller set of resrved inside-text-argument commands and, except for ESCAPE, these apply only to search text arguments and, with the ED command, these can be expressed in the CARET- (or UPARROW-) character form. CARET is a TECO–11 inside- and outside-text-argument command (but depends on the ET setting).

The most commonly used commands from this chapter are:

@ (command)/...text.../ , n ↑X , ↑X (inside), ↑N (inside)
↑Q (inside, TECO–11), ↑R (inside, TECO–1Ø), ^ or ↑ (TECO–11).

You should remember that the text delimiting character in @–modified text arguments is arbitrary (/ is common) but must not appear in the text.

You should certainly practise entering insertion and search commands in the @–modified form. You can include ESCAPE characters in this way and so create macros either through the text buffer or, in TECO–11, with the @↑U/...text... / command. You can now edit macros in text buffer as well.

Insert some text in the buffer and then execute a few indefinite searches through it using the ↑X and ↑N inside–text–argument commands. After this has been done store one of these commands in a Q–register and copy it into the text buffer to edit it. The task is to change the ↑X to a ↑N (or vice versa). This involves searching for the character with a literal specification command (what happens if you don't include the ↑Q , or ↑R in TECO–1Ø ?). A literal specification is also required to insert the new character in TECO–1Ø.

The other inside–text–argument commands can be applied when the need is perceived. There is not much point in practising the case conversion commands unless you will have to use them. Exact mode searching is very useful if you have files which make extensive use of both cases and a terminal with the full ASCII character set.

PART THREE

PROGRAMMED TECO EDITING

7. TECO as a Programming Language
8. Numeric Arguments
9. Extensions and Examples of TECO Programming.

This, the final part, extends to cover programmed TECO editing. This involves two groups of commands. The first comprises the TECO execution transfer commands; the second deals with manipulation of the numeric values which are used to form expressions which are tested to determine conditional execution structures. TECO is a powerful programming language but the structured approach as used here, is almost essential in writing understandable editing programs.

A knowledge of the commands described in the previous two parts is assumed. In particular the procedures of storage and subsequent execution of commands must be thoroughly understood.

This part completes the description of the TECO editing commands. Mastery, however, comes only with practice. No exercises have been prepared for this section; the best way to understand programmed TECO editing is to apply it to your own editing problems.

7. TECO AS A PROGRAMMING LANGUAGE

The information given to this point is sufficient for most interactive TECO editing. That is, editing in which the user makes the decisions as to what will be changed while the job is in progress. If your main interest is the correction of program source files this may be sufficient. However, there is another class of problem for which this approach is not convenient. An example of this type is the formatting of text. The text could be a book or it could be a source program. The characteristic which place this task in the class under consideration are that a large number of changes have to be made, the sequence of text changes required to produce the final format varies from, say, line to line and that the final format may vary from line to line.

Despite the fact that the precise sequence of changes required at any point cannot be stated explicitly in such a problem it is possible to develop an algorithm which describes how the desired format can be achieved. The algorithmic structures used in such a development may be the familiar sequential operations, "if-then-else", conditional execution, "repeat-while" or "repeat-until" repetitions and, in a limited way, the unconditional transfer of execution or "go to". The additional features of TECO which will now be described allow such algorithms to be implemented. In other words, complex editing tasks can thus be accomplished by writing a TECO *program.*

Many people are rather surprised when informed that programs can be written using TECO *as the language.* In fact every string of commands, e.g. JHT, is a program of instructions which describes the sequence of operation to be performed on the text being edited.

Perhaps this example is too simple to be classed as a program. However already you can write a TECO "program" to make a (simple) change to a whole file without intervention from the user. The command *J<FNX($)Y($);>($)($) changes every X in a file to a Y.

The procedure used in this case can be expressed as an algorithm:

```
move pointed to the start of buffer
store current value of automatic type-out setting
set for no automatic type-out

repeat
        repeat
                search for next X
                delete it
                insert a Y
        until   end of buffer reached
        fin
        output the page
        clear the buffer
        read a new page
until   no more input
fin
restore original automatic type-out setting
```

The use of the word "fin" (short for "finish") to point out the end of a structural block is somewhat unusual but is introduced here because it is a necessary indicator in the general TECO implementation of such units. The word "end" already has a well established meaning and has therefore been avoided for this purpose.

The angle bracket construction provides a sophisticated means of controlling TECO execution. It will be recalled that if the angle brackets are opened with a preceding positive numeric argument they indicate that number of repetitions of the enclosed group of commands. If the preceding numeric argument is negative the enclosed commands are skipped; that is, the brackets indicate conditional transfer of execution. If there is no preceding numeric argument control must be transferred out of the loop by the semicolon command. This has a more general function than has been explained so far. In general the semicolon command, which can be used only inside angle brackets, is written:

n;

and when the numeric argument is negative the semicolon is ignored. When the numeric argument is greater than or equal to zero the semicolon effects immediate transfer of execution to the command following the *corresponding* closing angle bracket.

It is possible to build up structured TECO programs using these forms. However, it is not convenient to do so in general. The depth of nesting of angle brackets is limited and there are some problems in developing a visually clear format which reveals the program structure. In practice most users apply these commands to fairly simple and readily comprehensible commands. TECO possesses an alternative set of execution control commands which, although of a rather lower level, are quite general. These will be used to implement structured TECO programming.

It is not suggested that "standard" high level commands, familiar to all TECO users, such as <FSstring 1 (Ⓢ) string 2 (Ⓢ) ;> should be broken into their underlying structural components. The aim of structured programming is to make a program easier to write correctly, to verify and to modify. This should be kept in mind. However, the TECO conditional execution commands allow the most general editing algorithm to be programmed.

That was the good news. The not-so-good news is that TECO, like FORTRAN, possesses the rather weak "if-then" and "go-to" constructions. This means that the favoured "if-then-else", "repeat-while" and "repeat-until" structures cannot be programmed directly in

TECO. There are two ways this can be overcome. The first is to follow a strict format in writing TECO. This approach will be followed in these notes. The second is to write a pseudo-TECO program using such words as "if", "then" and "else" and then edit this "High Level TECO" program (using another TECO program) to convert it to real TECO. This will not be pursued further but is worth thinking about.

You are most strongly advised to adopt one of these procedures. Do not write "unstructured" TECO; it is too hard to follow.

TECO has provision for non-executable comments within commands. These must be used freely. In fact, in the following sections the algorithmic key words, "if", "then", etc., are included as comments in TECO programs to accentuate the structure. An uncommented, let alone unstructured, TECO program is almost impossible to understand.

Finally, a TECO program can even be made interactive, to a limited extent. Messages can be typed out during program execution and execution may be stopped so that user instructions may be input to change the course of the editing.

When should you write a TECO program?

The answer to this question depends on your skill and experience with TECO. As a "rule of thumb", if you have to use pencil and paper to work out how to achieve the desired editing you should consider writing a program. If, as well as using pencil and paper to plan the commands, you will want to do the same editing job in the future you certainly should write a *fully commented* program and store it either on paper or as a file (ready for re-use) as appropriate.

You should remember that it is (at least) as easy in TECO as in any other programming language to make mistakes in logic, in syntax and in writing. This is why structured programming concepts can be so usefully applied in TECO. Additionally it will be seen that TECO possesses only limited aids to program error diagnosis.

Storing and using TECO programs

To create a TECO program you must first construct an algorithm which describes the sequence of operations required to achieve the editing task. Then you implement this (on paper) as a TECO program using the structured forms suggested. The next job is to enter this program as actual commands. The immediate aim is to store the program in a Q-register so that it can be executed as a macro. This is best done by entering the program as commands (that is, after the asterisk prompt). The CONTROL characters and ESCAPES complicate the alternative procedure of inserting the program into the buffer as text.

After a reasonable number of lines have been entered into the command buffer abort the command (↑G ↑G) and enter it into Q-register i with the command *i. Enter the next segment into another Q-register and finally copy the segments into the buffer to merge them in the correct sequence before returning the complete Q-register of a Q-register. At this stage (or earlier if it is a long program) you should output the program to a file in case of disaster. It is conventional to give TECO program files the extension .TEC .

After the program is verified (some useful commands to aid this are given later) a copy of the correct program should be made. This can used again whenever the same editing task is to be performed. The basic procedure is as follows:

_R TECO <cr> (for example)

*ERPROGI.TEC($)($)

*HKA Xi ($)($)

(reads the program and stores it in Q-register i)

*ERPROG.FOR($)HKA($)($)

(reads the file to be edited)

* Mi($)($)

(execute the program)

*EX($)($)

_

The Q-registers provide subroutine (procedures) capability in TECO programming. Frequently used subroutines can be stored in separate files and entered into the appropriate Q-registers as shown in the example above. Alternatively, the main-and sub-programs can be stored in one file and the program can be written so that it loads the sub-programs into the appropriate Q-registers automatically. This means that you can maintain a set of your own complex editing programs. For example you might have one to format your TECO programs. It cannot be over emphasized that you must make stored TECO programs as fully commented (and even self-documented) as possible. You will see why if you ever wish to modify one which isn't so written.

The terminal keyboard represents another input stream. TECO-11 under RSTS/E and RSX-11 allows this channel to be accessed by a file. The command:

EIfilespec($) (for Enable Indirect input)

presets TECO so that the next TECO call for terminal input will come from specified file and not from the keyboard. This is quite useful for loading TECO programs into Q-registers. The file containing the program is referenced in an EI command and when the asterisk prompt appears this file is loaded into the command buffer. It can be executed directly by typing double ESCAPE or entered into Q-register i by typing double BELL ((↑G) (↑G)) followed by *i. This can be done without disturbing the current input file.

Input from the EI file stops when the end of the file is reached. The first input goes to the command buffer and it can be executed as a command either by a typed double ESCAPE at the end of the file or by double ESCAPE in the file. In the latter case the remainder of the file is then available to be input at the next request for terminal input. If the executed commands include EI($) the file is closed and the remaining parts are discarded. This is significant under RSX-11 (see Reference Manual).

Any error which occurs after the EI command and before the call for input from the terminal will close the indirect command file. The file can be closed explicitly with the command EI($) . Since the files read by the EI command are usually TECO programs and are given the identifying extension .TEC this extension is assumed if none is specified. This is the only example of a default extension being assumed in TECO.

The input from an EI file is never combined with commands already entered through the terminal; the EI command simply presets where the next input will come from. It is not possible to insert the contents of an EI file into the buffer directly by typing I and then EIfilespec($) . To enter the file into the buffer it should be entered via a Q-register as described above. Keyboard commands can of course be appended to the end of the input EI file before execution is started.

TECO comments

Strings of text, enclosed by exclamation marks, may be inserted anywhere in a TECO command string (with the exception of text arguments) without effect. Such a string is used to include explanatory comments in a TECO program. Space and carriage return/

line feed are also ignored and should be used freely to document the program. For example a TECO program could start with a title as follows:

!PROGRAM TO EDIT TEXT!<cr>

!EDITED TEXT IS OUTPUT IN 1Ø LINE PAGES!<cr>

(commands follow).

Comments may be included in command strings and may separate numeric arguments from the commands to which they apply. They must not however separate a command which takes a text argument from the intended text because they will be interpreted as part of the text.

The TECO execution transfer commands

As mentioned above there are two execution transfer commands in TECO; unconditional transfer or "go to" and the conditional transfer or "if then".

The unconditional transfer command transfers execution to a particular location in the command string (or macro) marked by a label ("tag") which is simply a TECO comment (but limited to a maximum length of 128 characters). The tag referenced in this command, which has the form Otag(\$) or, in TECO-11, @O/tag/ (which is usually easier to read) must occur in the command string or macro in which the transfer command is given and its delimiting exclamation marks are not included in the reference. After transfer execution continues with the first command after the concluding ! of the tag matched in exact-case mode.

When an Otag(\$) command is encountered TECO searches that command string or macro level for the *first* occurrence of the referenced tag. The search commences at the start of that command string and thus the tag can appear either before or after the transfer command. As far as TECO is concerned it doesn't matter how many times the tag appears in that macro level; it is an essential requirement for clear programming that the referenced tag should appear once only in the command string or macro.

If you don't take care to make the reference unique you must expect the trouble which will surely befall you!

The following example illustrates the use of this command.

* (commands) J

ONEXT PART(\$)

(more commands)

!NEXT PART! T(\$)(\$)

After the first commands are executed the pointer is moved to the start of the buffer by the J command and then execution passes to the first command after !NEXT PART! (which is is T) and types the first line.

If the structured programming approach is to be followed, and this is strongly recommended, this command should be used with great restraint. In fact it is urged that you should, as far as possible, use it only in conjunction with the TECO conditional transfer command.

The conditional transfer command has the form:

(n)"*x* commands'

in which n is the numeric argument whose value determines whether a test, represented by a single letter (generally, *x*) succeeds or fails. If the test is satisfied then the commands which are

entered between the letter *x*, following the quotation mark ("), and the final apostrophe (' or ') are executed. If the test is *not* satisfied the commands enclosed within "*x* and the final apostrophe are skipped and execution passes directly to first command after the apostrophe.

The "*x* and the final ' must be used in matching pairs. They may be nested (like parentheses in arithmetic expressions), but must be complete within a single macro level. The closing apostrophe is a command termination; a numeric value, for example, cannot be passed across it.

The four basic test commands, represented above by *x*, are:

Letter:	*Execute the enclosed commands if:*
G	n is Greater than zero
L	n is Less than zero
E	n is Equal to zero
N	n is Not equal to zero

A number of other tests will be described later.

As is the case with unconditional transfer command uninhibited use of this conditional execution command is extremely undesirable. Recommended usages are described below.

You are explicitly warned against using these commands to transfer into or out of angle bracket execution loops. There is a way of transferring out with these commands but it is unnecessary; the semicolon command is the appropriate command for use with angle brackets.

Recommended TECO programming structures

The favoured command forms used in structured programming are the if-then-else conditional execution, repeat-while pre-tested iteration and the repeat-until post-tested iteration. These can be implemented in TECO but not as directly as we might wish. The main convention which is followed is that, in implementing the structured units, the command string which may or may not be executed depending on the result of a TECO conditional execution transfer test is always replaced by an unconditional transfer command. This is illustrated in the following preferred formulations:

If-then-else structure

A specific case will be shown first:

```
!IF     (n – 3) NOT greater than zero!  (n – 3)"G OELSE ($) '
!THEN!  (commands executed if n is not greater than 3)
     OFIN ($)
!ELSE!  (commands executed if n is greater than 3)
!FIN!
```

The first thing to notice is that the structure is expressed in the form "if something is *not* true-then-else". This is necessary to maintain the if-then-else flow of execution and, based also on experience with structured FORTRAN, the slight inconvenience involved in writing the commands is worthwhile. Any test can be used to control the execution in this structure and for the general test, "x, the form is:

!IF n"x NOT satisfied! n"x OELSE ($) ' etc.

where n represents an expression.

The practice of including explanatory comments is highly recommended. The comments need not take the form shown above.

The sequence:

!IF next character is not COLON! (ØA–↑↑:)"E OELSE $ '

(the command ØA is described later) is clearer than:

!IF (ØA–↑↑:) NOT equal to zero! (ØA–↑↑:)"E OELSE $ '

There is only one test which should be expressed differently. It is rather ridiculous to write " if n is not equal to zero is not true-then-else" when the tests "E and "N are complementary (these are the only complementary tests in TECO). In this case it is better to express the test:

!IF n equals zero! n"N OELSE $ '

The next thing to consider is what happens when there are a number of such structures in a program. All will end in !FIN! but TECO transfers to the first tag after an O command. The solution here is to identify each structure by a number which is unique in that macro level. The first structure is number 1 the second 2 and so on (a bit like statement numbers in FORTRAN); it is not necessary that these numbers be sequential but they must be unique.

If-then-else structure

```
!IF n"x is NOT satisfied!  n"x OELSE N$ '
!THEN!
        (commands executed if n"x is not satisfied)
        OFIN N$
!ELSE N!
        (commands executed if n"x is satisfied)
!FIN N!
```

The indenting of the commands is deliberate and is discussed further in a later chapter under format. The aim is always for clarity; this should be kept in mind.

Repeat structures

The considerations discussed above apply to the "repeat-while" and "repeat-until" structures which are given below in general form:

Repeat-while structure

```
!REPEAT N!
!WHILE n"x NOT satisfied! n"x OFIN N$ '
        (commands executed while n"x not satisfied)
        OREPEAT N$
!FIN N!
```

Repeat-Until structure

```
!REPEAT N!
        (commands executed at least once and until
        n"x is satisfied)
!UNTIL n"x IS satisfied! n"x OFIN N$ '
        OREPEAT N$
!FIN N!
```

Note that the UNTIL condition is not of the logical complement type. This has been chosen deliberately so that all the "repeat" structures (there is another) end of the same way.

The selection of these forms for recommendation has been made carefully and in the light of experience gained in writing structured FORTRAN programs. The reward for the restraint used in writing this way is in clearly, easily corrected and modified programs which, with the addition of a few explanatory comments are virtually self documented.

Examples of these structures are given in subsequent sections. In particular comments may usefully be included within these structures to clarify the test condition which is being applied.

A particular advantage of these structures is that they allow very direct implementation if the program algorithms developed in the "if-then-else" and "repeat" structural forms; the use of these key words to double as comments and tags is important. You can program these using angle brackets but comments tend to be left out as inessential and the program gets hard to follow. The author's preference is to limit angle bracket structures to sequences containing relatively few commands (maybe a dozen), written on one line and not nested. There's nothing really wrong with them; it's that they don't impose a clear programming style and in TECO programs clarity is a must.

Sometimes, after a formally written TECO program has been completed, it will be apparent that some structures within it can be expressed rather neatly with high level TECO commands. In principle there is nothing wrong with changing these if it makes you happier. However, this should not be done until the program is fully verified; it is terribly easy to lose the structured form when a program written with angle bracket commands is being patched up. It is not a good idea to invoke a macro from within an angle bracket loop; it is too easy to lose control over the limited depth of nesting allowed.

Direct entry of TECO programs

TECO-11 programs may be invoked directly from the system. Under RSTS/E the system command

MUNG filespec *or* MUNG filespec,text

is equivalent to

RUN $TECO

*Itext(ESC)EIfilespec(ESC)(ESC)

That is, the (optional) text is inserted into the buffer and the named file is specified as an indirect file entered in response to the request for terminal input.

Under RSX-11 the corresponding commands have the same form:

MUNG filespec *or* MUNG filespec,text

equivalent to

TECO

*Itext(ESC)EIfilespec(ESC)(ESC)

If the MUNG command is not available the system command

TECO @filespec

is equivalent to

TECO

*EIfilespec(ESC)(ESC)

Under RT-11 TECO programs can be *executed* directly from the system level. The command

.EDIT/EXECUTE[:text] filespec

is equivalent to

.R TECO

*ERfilespec$HKA HXZ HK Itext$MZ$$

That is, it reads the first page of the named file (this file remains enabled for further read commands) and copies it to Q-register Z. It then clears the buffer and inserts the specified text and finally executed the text in Q-register Z as a macro.

8. NUMERIC ARGUMENTS

Numeric arguments are used to control all TECO conditional execution commands. In this chapter the provisions of TECO for generation and storage of numerics are explained. Particular attention must be given to the rules for forming numeric expressions; these have a number of unconventional features.

Numeric expressions

Numeric expressions may be formed from numbers entered in a command string and from commands which return numeric values. The complete set of rules for forming these expressions will now be discussed.

A numeric expression is only significant if it precedes a command which makes use of a numeric argument. If it is not followed by such a command it is evaluated but has no effect. On the other hand, if it is important that a numeric expression should be evaluated but should not apply to the following command (e.g. the % i command, explained later in this chapter) then an ESCAPE character entered after the numeric expression will stop it from acting as an argument.

Numeric expressions are formed by combining numeric terms with numeric operators. The arithmetic operators defined in TECO are +, –, * (multiplication) and / (integer division). These have their normal meanings, but recall that in integer division the remainder is lost. A + before an expression is ignored and a – before an expression negates it, that is, it is equivalent to multiplying the expression by –1. In TECO-1Ø a SPACE in an expression is equivalent to +.

In TECO it is not illegal to combine numeric terms without operators. In this case only the last numeric term in the unseparated string has an effect, except that an operator preceding the string continues to apply. Thus:

B.Z=Z
–Z53=–53
8–ZZ.=8–.

It is important to realize that such numeric strings are not treated as errors in TECO; you may well have left out the operator by mistake but TECO will not indicate this.

Normal numeric expressions with operators are performed from left to right. There is *no hierarchy* of operators That is, the expression:

3+4*5 =35 (not 23).

It is fortunate that simple addition and subtraction is more common in TECO than multiplication and division but when you do use the latter you must be very careful to check that expressions are evaluated as you intend. Parentheses must be used freely to make clear the order of evaluation; all expressions inside parentheses are evaluated before operators outside parentheses are performed, Thus:

3+(4*5) =23 (as expected).

The expression:

–(3+(5*(7/(–2)))) =12

and it will be seen below that the parentheses around the term –2 are essential.

Two other operators find use in forming TECO expressions. These are the logical operators & (bitwise logical AND) and # (bitwise logical OR). The characters & and # are known as ampersand and hachure (hash) respectively. Bitwise logical operators work by examining the binary representation of the numbers being operated on. The AND operator generates a new number which has a 1 in every bit position in which *both* input numbers have a 1 and the OR operator generates a new number which has a 1 in every position where *either* of the input numbers has a 1. All remaining bit positions are set to Ø.

In practice these operators are used very simply. The binary representation of Ø has every bit position occupied by a Ø and the representation of −1 has each position occupied by a 1. Thus:

(−1) # (−1) = (−1)
(−1) & (−1) = (−1)
(−1) # Ø = (−1)
(−1) & Ø = Ø
Ø # Ø = Ø
Ø & Ø = Ø

These results conform to the convention that (−1) is logical TRUE and Ø is logical FALSE. However in TECO there are no defined logical variables and you may find it preferable to think in terms of (−1) and Ø as "all bits 1" and "all bits Ø" respectively rather than as TRUE and FALSE. The use of parentheses , (−1), may seem excessively cautious; it is not, as will be shown below.

There are other things you *could* do with logical operators. For example you could form expressions using 1 instead of (−1). It isn't worth while. It is far safer to stay with one completely conventional usage. One of the very rare occasions in TECO in which logical operators are used in another way is in the setting of the bit-encoded ET word as described without explanation previously.

If you stick to the recommended usage with (−1) and Ø, the logical complement can be formed quite simply. If n is a logical value, (−1) or Ø, the expression:

−(n + 1)

gives the complement. That is, if n=(−1) this expression is equal to Ø and vice versa. In TECO-11 this can be formed more directly with the ↑_ (or CARET/UNDERSCORE or BACKARROW) command:

n ↑_ which is equal to −(n + 1).

This is still evaluated if n is not equal to (−1) or Ø but its usefulness in such a case is not obvious.

The reason for enclosing the (−1) values in parentheses in these examples has to do with the way in which TECO handles adjacent numeric operators. Adjacent operators occur in the expression:

Ø & −1

but not in

Ø & (−1).

The difficulty that this raises is that adjacent operators are not defined mathematically but, at the same time, TECO does not recognise them as errors. Unfortunately TECO-1Ø and TECO-11 handle adjacent operators very differently.

In TECO-11, when adjacent operators are encountered the whole of the expression to the left of the second operator is discarded. The second operator is taken to be the start of a new expression. Thus:

Ø & −1 = −1 and 3 + −−4 = −4

while

Ø & (−1) = Ø and 3 + (−4) = −1

There is of course no problem in an expression such as:

3 + ES

even if the command ES should return the value (−1) since here there are no adjacent operators in the command string. The negative sign is incorporated in the internal representation of the number.

In TECO-1Ø the treatment of adjacent operators is quite odd. If adjacent operators are encountered the last numeric value encountered is interpolated between the operators. Thus:

−2−+3 = −2−(2)+3 = −1

Clearly you must take care to avoid such expressions. The particular problem in TECO–1Ø is that a SPACE in an expression is equivalent to a + ; this has strange consequences. For example, the expression:

4 + 7

may be written with SPACE characters to improve legibility. Unfortunately it is equivalent to:

4+4+4+7 =19

and is not equal to 11. The combination SPACE+SPACE is taken to be +++ and the last numeric (4 in this case) is interpolated between each adjacent pair.

This is why it was recommended that SPACE characters should not be used in TECO–1Ø expressions. They should also be avoided in TECO–11 lest you develop bad habits which could cause disasters when you use TECO–1Ø.

The value of a numeric expression may be typed out by using the expression as the numeric argument of the equals command:

n=

As explained earlier, this command causes the value of n to be typed out on the terminal followed by CARRIAGE RETURN and LINE FEED. This command can be modified so that the carriage is not moved after the number is typed out by entering it as follows:

n:=

This option only applies in TECO–11.

Encoding and Decoding

In editing the need may arise to write a numeric argument as text. For example, if pages are counted automatically as editing proceeds you may wish to enter the page number as text at the top or bottom of the buffer. Alternatively, a page number may be included in the text and you may wish to use this number as a numeric argument.

The act of expressing a number as a string of ASCII characters is called decoding. That is, when the *single* number which is stored in the computer core in binary *code* as, for example, 11Ø1ØØ1 is expressed by writing the *three* characters 1, Ø and 5 (which we read as 1Ø5) it has been decoded. The reverse procedure in which the three characters 1,Ø and 5 converted to a single number in the computer is called encoding. In TECO these operations take the form of transfers between numeric arguments and the text buffer.

These transfers between text and numeric argument can be achieved with two forms of the Backslash, \ , command. The command n\ transfers (decodes) a numeric argument n to the text buffer. The decimal expression of the value of n is inserted as ASCII characters immediately to the left of the current pointer position. That is, if the numeric argument n is equal to 437 the ASCII characters 4, 3 and 7 are inserted before the pointer.

The reverse transfer (encoding), from text to numeric argument, is achieved by the command , \ , (with no numeric argument). Note that the \ command is therefore a numeric argument itself and is used to precede a command which requires such an argument. The exact function of this is to take on the decimal value of the string of digits (optionally preceded by a + or – sign), which starts immediately after the current pointer position and is ended by the first *nondigit* character encountered, and to *move the pointer* to the end of this string. If there is no string the value Ø is returned.

Thus if the buffer contains the following section of text . . . ABC345 61Ø <space> 2EF. . .
↑
the command returns the numeric value 61Ø for use as an argument and moves the pointer position between the Ø and the space which precedes the 2.

Q-Register Numeric Storage

The essential features of the Q-registers have been described previously in connection with the Xi and Gi commands for text storage and the *i and Mi commands for command

storage. The 36 Q-registers are identified by the letters A–Z and numerals Ø–9.

The Q-registers can also be used for the storage of integer numeric arguments. The command nUi (for Update) transfers the numeric argument n to Q-register i *overwriting* any previous contents. Note that the argument is *transferred* and cannot be used as an argument until it has been retrieved. For example, the command 2UA= is not valid.

To copy the contents of Q-register i for *use as a numeric argument* (not as text) the command Qi is used. That is, the command Qi returns the numeric value stored in Q-register i as a numeric argument. Thus the correct form of the invalid command given above would be

*2UAQA= ($)($)

2

*

In this command string the integer 2 was stored in Q-register A by the command UA and retrieved for use as the numeric argument of the = command by the command QA. The contents of Q-register i is not changed by the command Qi.

The integer stored in Q-register i can be incremented by 1 and the *incremented* value returned as a numeric argument for the next command with the command:

%i

This command is equivalent to the command string:

(Qi + 1)UiQi

If the stored number is to be incremented but it is not wanted as an argument the command should be terminated with a single ESCAPE character, as described previously.

In TECO–1Ø a stored integer can be incremented by n using angle brackets (but the value is not returned because the closing angle bracket is a command terminator):

n<%i>

In TECO–11 the angle brackets are not required; the command:

n%i

increments the integer stored in Q-register i by n and returns the incremented value.

To return the contents of Q-register i as *text* the n\ command must be used. Thus, to enter the integer stored in Q-register A as ASCII characters immediately to the left of the pointer the following command would be given

*QA\ ($)($)

The Colon Command

The colon (:) is used extensively in TECO–11 but in various ways. In one use it signifies a type-out action, for example, :G* which types the last filespec on the terminal or := which modifies the = command so that the carriage is not moved after the numeric value is typed.

The most general use of the colon is to modify a command which may or may not produce an error condition depending on the state of the command environment (the buffers, pointer position, stored files etc.). A command of this type, when preceded by a colon, returns a numeric value as well as performing its normal function. If the command can be executed successfully the value returned is (–1); if it fails the value returned is Ø and any error procedures (e.g. typing of error messages and return to the command input state with the asterisk prompt) are skipped and execution passes to the next command in the string.

In TECO–1Ø the sole use of the colon is in one particular example of the general type. Any search command (S,FS, N,FN, UNDERSCORE or BACKARROW) can be preceded by a colon to modify its behaviour as described. If the search is successful the command behaves normally but also returns the value (–1); if it fails there is no error message and the

value ∅ is returned to act as an argument for the next command. The pointer is, however, still moved to the start of the buffer in the event of a search failure. The colon is entered after any numeric argument but before any @ modifier, thus:

:S...text...$ or n:S...text...$ or n:@S/...text.../ .

This also applies in TECO-11 except that if the colon modified search command precedes a command which requires a positive argument (e.g. the nP command) a value of (−1) is interpreted as 65535. Care is needed.

In both versions of TECO the most common use of this modified command is in controlling an indefinite repetition and, recognizing this, *all* search commands inside angle brackets are automatically colon-modified. This is why they do not generate error messages when they fail. This also explains why a search command inside angle brackets and followed by a semicolon controls the execution. The (automatically) colon modified search acts as the numeric argument for the n; loop break command.

The TECO-11 bounded search command can be used to produce a special form:

(1,1)S...text...$

Which acts as a comparison command. If this is expressed as a colon-modified search:

(1,1):S...text...$

then, if the text which lies immediately after the pointer matches the specified text argument the pointer is moved to the end of the matched string and the command returns the value (−1). If a match is not obtained the pointer is not moved and the command returns the value ∅. This is a useful command and it can be expressed in the special short form:

::S...text...$

In TECO-11 the colon modifier can also be used to precede the file opening commands ER, EB and EN. If the specified file exists the value (−1) is returned; if it cannot be found (this would normally be a fatal error) it returns the value ∅. This can be useful in programmed editing where input is to come from one or more of a number of files which may or may not be present.

Another use in TECO-11 is in connection with the Q-register pushdown list described in the next chapter.

More Commands which Return Values

It is often useful in programmed TECO to control the sequence of command execution according to the text in the buffer. In TECO-11 the ::S comparison may sometimes be used for this. Another way, available in both TECO-1∅ and TECO-11, is with the command:

nA (for ASCII value)

This is distinguished from the append command by the numeric argument (which must be present); the append command, A, does not take an argument.

In TECO-11 the command nA returns the value of the (decimal) ASCII code of the $(.+n+1)^{th}$ character in the text buffer. The commonest form of this command is:

∅A

which returns the value of the next character after the pointer. In TECO-1∅ this command refers only to the character after the pointer regardless of the value of n; for consistency the form ∅A should always be used.

In TECO-1∅, if the pointer is at the end of the buffer this command returns the value zero. In TECO-11 the command nA where n is greater than or equal to zero and where the pointer is at the end of the buffer is regarded as an attempt to move the pointer off the page and is an error. An analogous situation applies at the start of the buffer. This command should not be used to see if the pointer is at the end of the buffer. The correct command for this is:

(Z−.)

which returns zero if the pointer is at the end.

The system-dependent commands (↑F) and ↑H (enter as CARET-H because CONTROL/H is BACKSPACE) return the current value of the console switch register and the time of day respectively and (↑B) (TECO-11) returns the date. In TECO-11 (↑V) returns the version number of the TECO program; this book is based on version 27.

In TECO-1Ø a specific use is made of the version number. From time to time changes are made to the TECO program. Some of these can affect the operation of programs written in accord with previous versions. For example, the P command in early versions created a <formfeed> character; this is no longer the case. The TECO EO (Enable Old) command changes the current TECO version to the specified old version so that old programs can be successfully operated.

The default option is that TECO is set for the latest version. The command nEO (n>Ø) sets TECO to operate the specific version which is required. For versions up to version 21, n is equal to 1. Versions 22, 23 and 23B the value of n is 2. The default (newest) version can be restored by the command ØEO (or nEO with n negative). The command EO with no argument returns the current value (n) of the EO flag.

In order to protect a TECO-1Ø program from future changes it could have as one of its first commands, nEO where n corresponds to the version for which it is written (currently 23B, n = 2). Its last commands should include ØEO to restore the default version setting. This command is not used in TECO-11.

Compound Logical Expressions

It is frequently convenient to base a conditional execution command on a compound logical expression. For example, an alteration to the text in the buffer may be programmed as a "repeat until" structure in which repetition is terminated when *either* all the text has been treated (−1 placed in Q-register A, otherwise Ø) or an error condition has been met (−1 placed in Q-register B, otherwise Ø). This could be expressed:

```
!REPEAT!
        (editing commands)
!UNTIL! (QA#QB) "L OFIN $ '
        OREPEAT $
!FIN!
```

The important thing about such compound logical expressions is that, even when properly used, they are mainly a convenience to the programmer. Subject to some qualification, the computer "doesn't care" whether it has to execute a compound logical expression or, instead, a sequence of simple conditional execution commands. The second approach is always open to the programmer; the compound logical expressions are simply an alternative. When compound logical expressions are not properly used they are positively harmful to good programming.

Compound logical expressions are properly used when they can be clearly understood (by human readers) and when they significantly simplify the appearance of a program (again, to a human reader). Even if these conditions are met, if the time taken to construct and write the compound forms is significantly greater than that required to code the simple sequential alternative form then the use of the compound form, being uneconomic, is improper.

A string of logical expressions all separated by "or" or "and" as in:

if (condition A) *or* (condition B) *or* (..... etc) is true then ....

can be readily understood. Similar strings preceded by a simple negation (of the whole string) are also readily grasped.

However, as soon as "and" and "or" are used in the one string, (and parentheses become necessary to make the meaning clear) the risk of misunderstanding greatly increases.

Add a couple of "not" operations and the expression runs the risk of being effectively incomprehensible. Such expressions have no place in any sort of programming.

To these general cautions must be added the consideration that TECO does not possess a simple command structure which allows direct use of compound testing such as:

if ((A = B) or (C ≠ D)) and (E < F)) then ....

where A through F represent numeric arguments.

This is, in the light of the comments above, just about as complex a logical expression as may be usefully employed and in TECO programming, in particular, is certainly complex enough to require the exercise of some thought in its implementation. The clearest procedure is probably the one shown below (referring to the above example).

```
!IF!    ØUA  ØUB  ØUC
        (A–B)"E –1UA'
        (C–D)"N–1UB'
        (E– F)"G–1UC'
       ((GA#GB) &GC)"E OELSE (Ⓢ) '
!THEN! (commands executed if expression equal to –1)
          etc.
```

The block of conditional execution commands used in this example do not themselves conform to the previously recommended structured forms but in this case clarity, one of the aims of structured programming, is better served by the simplicity of the scheme demonstrated here.

A particular trap in using logical operators lies in expressions in which the terms can have values other than Ø or (–1). For example, it is correct to use the following form to execute certain commands only if the character after the pointer is A or F.

```
!IF! ØU1 ØU2
     (ØA – ↑↑A)"E – 1U1'
     (ØA – ↑↑F)"E – 1U2'
      Q1#Q2+1"N OELSE'
!THEN! (commands executed if next character is A or F)
          etc.
```

However it is *not* correct to use the following sequence:

```
!IF!  (ØA – ↑↑A – 1)#(ØA – ↑↑F – 1) + 1"N O ELSE'
              etc.
```

This will work if the next character is A or F but it will also result in the execution of the commands in question if the next character is E. Logical operators are tricky things.

There are lots of clever things you can do with logical operators but the cleverest thing of all is limit use to simple expressions with terms which are Ø or (–1).

As indicated previously, TECO includes a number of other test commands which can be specified by writing the appropriate letter in place of the *x* in the general form (n)"*x* (commands)'.

Thus the test will be satisfied, and the commands delimited by "*x* and' will execute if, with *x* replaced by:

C n is the decimal ASCII code of an alphabetic Character (upper or lower case letter) a digit, $ sign or . (period or full stop);

D n is the decimal ASCII code of a Digit;

A n is the decimal ASCII code of an Alphabetic character (i.e. upper or lower case letter);

In TECO-1Ø the "C test is satisfied by a % sign in addition to the others listed (alphanumeric, $ or .). There are two other tests:

V n is the decimal ASCII code of a lower case letter.

W n is the decimal ASCII code of an upper case letter.

In TECO-11 it is hard to believe but the "V and "W tests have reversed meanings:

V n is the decimal ASCII code of an *upper* case letter.

W n is the decimal ASCII code of a *lower* case letter.

In addition, TECO-11 allows another test:

R n is the decimal ASCII code of an alphanumeric (upper or lower case letter or a digit).

In both versions the letter L can be replaced by T (for True) or S (for Successful search) while E can be replaced by F (for False) or U (for Unsuccessful search). These are merely a mnemonic convenience. In fact they can be dangerous if used unthinkingly. The two commands T and S are satisfied by an argument which is merely negative and not necessarily equal to −1 (logical true).

TECO-11 also allows the substitution of "> in place of "G and "< in place of L. A choice of so many forms when one will do seems excessive (even for TECO). You only need to remember "G, "L and "E.

Decimal and Octal Integers

To this point it has been implied that all TECO numeric operations assume ordinary (decimal) integers. This is indeed the case and the main difference between TECO-1Ø and TECO-11 in this regard is that in the former the range of integers allowed is from -2^{35} to $2^{35}-1$ (−34 359 738 368 to 34 359 738 367) while in TECO-11 the range is from -2^{15} to $2^{15}-1$ (−32768 to 32767).

These ranges reflect the internal representation of the numbers in binary form. Thus, in TECO-11, 16 bits are available per interger and one of these is reserved to indicate the sign. This may possibly be a significant consideration in some cases because the maximum integer size in TECO-11, while more than adequate for any normal editing task, could conceivably be exceeded. Up to the limit of 65 555 an unsigned number greater than 32 767 is treated as a negative number. For example, the command

65535=

types out −1. These problems would surely never arise with TECO-1Ø.

TECO-1Ø has limited facilities for handling octal (base eight) integers. The command ↑O (CARET/O, *not* CONTROL-Q, for Octal) preceding a numeric entry from the *terminal* causes it to be interpreted as an octal integer. For example

↑O1Ø

is interpreted as decimal 8 and this value is used to evaluate the argument in the TECO command string. For example:

↑O1Ø+3 = 11

In TECO-11 the ↑O command is not limited to the immediately following numeric input only. Instead it applies to all subsequent numeric input until it is cancelled by the command ↑D (for Decimal) which restores the default condition.

All TECO calculations are based on decimal numbers; the only purpose of the ↑O command is to get octal *input* accepted in the decimal form used by TECO. It therefore follows that values returned by TECO commands and used in numeric arguments are not affected by ↑O commands; they are already in the correct form. Only new input is affected.

TECO-11 makes particular provision for octal encoding and decoding. Even if a ↑O command is current the command

n\

inserts the input value n into the text exactly as it is entered. That is, if 1Ø is entered the characters 1 and Ø are inserted (not 8). In the opposite operation of encoding text as a numeric, if a ↑O command is current the \ command returns the value of the ASCII digit string read as an octal number and converted to decimal form. The string 1Ø will be returned as 8 for use in numeric arguments.

When octal input is current the digits 8 and 9 are illegal. This applies to the n\ and \ commands in TECO-11.

The only call for the conversion of TECO's decimal numbers into octal form arises when the user would like to see the numbers in octal form; TECO never uses octal numbers for its operations. Accordingly the only decimal-to-octal conversion occurs within a type-out command. The double equals command:

n==

converts the decimal number n to octal and types it out on the terminal followed by CARRIAGE RETURN and LINE FEED. The command:

n:==

has a similar effect except that the carriage is not moved after the number is typed. No sign is involved in the conversion of a number to octal representation. Thus in TECO-11 if you had to convert the decimal number 65 535 to octal it would be typed out as 177777. You would never be concerned about this with TECO-1Ø. The double equals command has no effect on the prevailing mode of interpretation for input.

It may be useful to recall that if you should wish to know the octal ASCII code of a character, x, you can give the command:

(↑↑) x==

and it will be typed out.

TECO Variables

It will have been apparent from the examples given in this chapter that the capacity of the Q-registers to store numeric values can be used to give TECO a facility for handling variables. A numeric value can be assigned to a particular Q-register and recalled as required for use or for alteration and re-storing. The difference from high-level programming languages is that the variables are identified only by storage location and *not* by a unique and permanent name. This way of using storage registers will be familiar to the many users of programmable pocket calculators. The responsibility that it imposes on the user is that of ensuring that the register invoked actually contains the data appropriate to the wanted variable. In particular it is necessary to ensure that Q-register contents are not over-written by macros called from the main program. A way of guarding against this is described in the next chapter.

There is an error in the current version of TECO-11 which results in the error message "missing right parenthesis" when a macro which returns a numeric value is included (correctly, according to the rules) in a parenthetical expression. The user must allow for this. If necessary the value returned can be stored in Q-register i before the expression is evaluated and recalled with the command Qi within it.

9. EXTENSIONS AND EXAMPLES OF TECO PROGRAMMING

This chapter completes the description of the TECO commands. Additional structured programming units are described and some examples of TECO programs are given. The chapter ends with a discussion of the costs of using TECO.

Interactive programmed TECO

TECO can type out programmed messages during execution. This is in addition to any type-out of text or numeric values. The message which is to typed out is entered in the command string at the place where the message is wanted and is delimited by a ↑A character at each end of the message. The ↑A characters are not typed out. The message can contain any character except ↑A and may be of any length. The first ↑A (for ASCII string) character is the actual command and can therefor be entered in either the CONTROL-A or CARET/A form; the second is the message delimiter and as such is not a command and must be entered in the CONTROL-A form only.

TECO-11 allows the alternative text argument form:

@ ↑A / . . . message . . . / or @↑A / . . . message . . . /

where / represents an arbitrary character (not in the message). This avoids the need to include a second ↑A in the message argument and permits the user to maintain a convention of using the CARET-character form in TECO-11 command strings.

One use for this command is to identify type-out commands. For example, a program might type out the page number in one case and a line number in another. The very real risk of confusion is removed if the type-out command is preceded by a message such as:

↑A PAGE NUMBER = ↑A

Particular attention must be paid to the carriage position characters when using this command. For example the adjacent ↑A commands:

↑A . . message 1 . . ↑A ↑A . . message 2 . . ↑A

will type:

. . message 1 . . . . 2 . . (no separation of messages) unless SPACE and CARRIAGE RETURN (and LINE FEED) characters are explicitly included in the messages. Messages specified in this way but interspersed with type-out commands (T, =, := etc.) must be written to take into account the position of the carriage after each typing action.

A more complex but very useful application of this command is illustrated below in connection with the interactive input command.

When the command ↑T (for Type-in) is encountered in the execution of a command string execution pauses until a single character is entered on the terminal keyboard. When this has been entered execution starts again with the ASCII code number of the entered character supplied as the numeric argument for the next command in the string. Some characters may not be entered this way (e.g. ↑C in some systems); consult the Reference Manuals on this point.

A common example of the use of the ↑A and ↑T commands is in the interactive control of the execution sequence. This is illustrated in the following sequence:

```
↑A TYPE Y TO CONTINUE; ANYTHING ELSE WILL CAUSE EXECUTION TO STOP: ↑A
!IF! ( ↑T – ↑↑ Y)"N OELSE $ '
!THEN!
        (commands which continue execution)
        OFIN $
!ELSE!
        (commands which stop execution)
!FIN!
```

First the message telling the user what to do is typed, then execution pauses while a character is entered. The ASCII code of the entered character is tested to see if it is equal to the ASCII code of the letter Y and the result used to control the sequence of execution.

The ↑T command only accepts one character but this does not prevent strings of characters from being entered. A crude example follows; execution pauses while text is inserted into the buffer:

```
↑A ENTER TEXT TERMINATED BY AN ESCAPE CHARACTER
↑A
↑T UA
!REPEAT!
!WHILE NOT ESCAPE! (QA-↑↑S)"E OFIN $'
        QAI$    ↑T UA
        OREPEAT $
!FIN!
```

After the message is typed the execution pauses while a character is entered. Its value is stored in Q-register A. The stored value is then tested to see if an ESCAPE character was entered and if one was, execution passes on to the next part of the program. If the character was not an ESCAPE it is entered in the buffer to the left of the pointer, execution pauses for the next character to be entered and the testing process is repeated. It is assumed that nothing important was stored in Q-register A.

This is a crude example because it makes no provision for the correction of errors in the entry of the text from the keyboard. One way of overcoming this would be to type out the entered text and have the user type Y if it is correct. If it is not correct the entry procedure can be repeated until it is completed correctly. Another approach would be to allow RUBOUT or other immediate function characters to act in their normal way to delete characters from the string.

TECO-11 has a number of additional provisions for using the ↑T command. If Bit 3 (2^3 =8) of the ET word is set the characters entered through the keyboard will not be echoed as they are typed. An obvious case where this is desirable is in the entry of secret passwords. Bit 5 (2^5 = 32) of the ET word is also concerned with the ↑T command. If this bit has been set when execution reaches a ↑T command there is no pause for entry. If a character has already been entered it is accepted but if there is no character ready waiting for the ↑T command the value (−1) is returned and execution continues without pause.

In systems where the EI command is implemented the ↑T command can be used in a procedure such as that illustrated above to input text from the specified file rather than from the keyboard. This provides a way of avoiding the need to close and subsequently reopen the current input file while the contents of another file are being interpolated. Bit 3 of the ET word might well be set to no echo when such input is being accepted.

The Q-Register Pushdown List

The previous example of the ↑T command function made use of a particular Q-register to store the value of the entered character. It was assumed that nothing was stored in it beforehand. This is not at all desirable if a program (like the example) is to be stored in a file for reuse from time to time as a general purpose procedure (macro). This problem is overcome with the following TECO feature.

An additional 32 storage locations are available *during the execution* of a TECO command string. These locations are *cleared* every time TECO completes execution of a command string and types an asterisk. These storages are called the Q-register Pushdown List and the command [i (left square bracket i, read as "pushdown") copies the contents of Q-register i into the top register of the Pushdown List (without changing Q-register i). The maximum depth of the stack is 32 entries.

The command]i (right square bracket i, read as "pop up to i") transfers the contents of the top register of the Pushdown List to Q-register i, erasing it from the Pushdown List and overwriting the contents of Q-register i. The entry which was made in the Pushdown List immediately before the one which is transferred is now at the top of the Pushdown List. It is important to realise that once an entry has been made in the Pushdown List there is no way of telling from which Q-register it originally came. There is only one Pushdown List and it serves all Q-registers. Furthermore, the entries can be retrieved only in the reverse order of entry (last in - first out).

The contents of Q-registers can be exchanged using the Pushdown List. For example.
*[A[B]A]B (\$)
exchanges the contents of Q-registers A and B without affecting any other Q-registers. This can be extended to more complex re-arrangements. Another use when the strictly sequential order of storage could be used would be to reverse the order of the characters in a string of text.

However, the main use of the Pushdown List is in making TECO macros truly general purpose. A macro may be written using Q-registers A through D (for example). The problem is that these registers may already be in use; the solution is to start the macro with the command sequence:
[A [B [C [D
which puts the previously stored material safely into the Pushdown List and frees these registers for use in the macro. The macro then ends with the sequence:
]D]C]B]A
to restore the Q-registers to their former state.

The problem is a little more difficult if the function of the macro is to store material which is to be preserved after the execution of the macro has been completed. In this case care must be exercised to ensure that this does not overwrite other wanted material. Each program should have an introductory comment section which gives details of such requirements.

It is not only Q-register contents which should be preserved in a general purpose macro. If special flag settings (e.g. search mode, ET etc.) are required in the macro then the previous settings should be stored in Q-registers (or the Pushdown List) for the duration of the execution of the macro and be restored at the end.

The [i and]i commands in TECO-11 have a very useful feature; a numeric argument entered immediately preceding these commands continues to apply just as if they were not present at all. A numeric argument may be validly entered before an Mi command so that it applies to the first command in the macro. With this feature the TECO-11 macro may start with a series of push down commands and the calling numeric argument will apply to the next command after this series. Comments are totally ignored and have no effect in any case. The same applies at the end of a macro in TECO-11; the macro may generate a numeric value which is to apply to the first command after the Mi command and still end with a series of pop up commands. This does not work if the Pushdown List was empty when the pop up command was given.

This feature is not available in TECO-1Ø. A reasonable convention here is that one particular Q-register (Q-register Ø is suggested) should be reserved for temporary storage of numeric arguments passing between the command string and the macro. A macro with a calling numeric argument would then start:
UØ [A [B [C (etc.) QØ (first command . . .)
An analogous pattern would be used to pass a numeric out of the macro.

TECO-11 allows a nice procedure for aborting the execution of a program. Any TECO execution can be aborted by typing (↑C) (or (↑C) (↑C), system dependent). The trouble is that if the previous contents of Q-registers or various flag settings have been stored in the Pushdown List by the program they will not be returned and will be lost since the Pushdown List is cleared when execution ends. In TECO-11 this can be overcome using Bit 15 ($-2^{15} = -32768$) of the ET word (this bit sets the sign to "-"). If this bit is set (equal to 1) TECO-11 records that a (↑C) has been typed by turning off this bit (setting it equal to Ø) but does *not* terminate execution.

To make use of this feature the TECO-11 program is written as a "repeat while Bit 15 is set" structure:

```
!SAVE PREVIOUS STATE IN PUSHDOWN LIST!
[A [B [C [D     (etc.)
!SET ET BIT 15!
(-32768#ET)ET
!REPEAT 1ø!
        !WHILE ET BIT 15 SET!  ET"L OFIN 1ø($)'
        !THE ACTUAL PROGRAM GOES HERE!
        .....
        !AT THE END OF THE PROGRAM TURN OFF
        BIT 15 SO THAT THE PROGRAM WILL STOP!
        (32767&ET)ET
        OREPEAT 1ø ($)
!FIN 1ø!
!RESTORE THE PREVIOUS STATE!
]D ]C ]B ]A
```

(Note that a special technique is used to switch off Bit 15.)

This is such a useful procedure that it is probably worthwhile remembering it as one of the "standard" TECO-11 forms and writing it with the high level TECO conditional execution commands < n;> thus;

```
[A [B [C [D     (etc.)
(-32768#ET)ET<ET;
!THE ACTUAL PROGRAM GOES HERE!
(32767&ET)ET>
]D ]C ]B ]A
```

This can readily be included in any TECO-11 program which makes use of the Pushdown List.

Bit 6 (2^6=64) in the TECO-11 ET word can be used to allow a TECO program to run with the terminal detached. This is system-dependent (see Reference Manual). It requires a lot of confidence in the program before this option can be used. If this is used it is wise also to use the option controlled by Bit 7 of the ET word. If this bit is set any error condition encountered in a TECO program results in the termination of TECO after the error message is printed. This option also prevents the typing of informational messages and causes immediate termination of TECO when a (↑C) is typed. It is, however, no help if a detached program enters an infinite loop.

Correcting TECO programs

TECO possesses limited aids to program correction. When an invalid command is encountered the ? command given immediately will type out a section of the current command string up to the error but in TECO-11 this may involve an inconveniently large amount of typing if the program is large. The ? command can be used in another way. When the ? is not the first command after an error it is treated as a normal command and when execution reaches this point TECO procedes to type out each command as it is executed. This command tracing continues until another ? is encountered or, in TECO-11, until the end of the current command string is reached.

Again, this can produce an inconveniently large amount of type-out, though this is a matter for the user to decide. One problem with TECO-11 is that as the search for tags referenced in O commands proceeds from the start of the current macro level *all* comments/ tags are examined fully and are typed out under the ? command with no additional CARRIAGE RETURNs added. This produces a lot of type-out and, on some terminals, the carriage types away over at the right hand side of the page. This does not occur with TECO-1ø.

The principle involved in using the ? trace mode is that the user can follow the progress of execution and can trace the flow through various conditional paths. It is often more convenient in practice to achieve the same thing by including a number of ↑A messages in the program. These might be a simple series of numbers or letters or quite detailed descriptions of the location. The purpose is to show that execution has passed a certain point.

It is not possible to have TECO pause in the execution of a macro but it is often desirable to see how the buffer and pointer position have been changed by a series of commands. One convenient way of doing this is to include a sequence such as ØT HT T, identified with a type-out message, at key points. Important Q-registers can also be examined in a similar way. Remember that if you use the Pushdown List you will not be able to examine the actual Q-register contents (as they were during execution) after the program execution has finished. The push down and pop up commands which make the program general-purpose should therefore be added after all corrections have been made when the temporary type-out commands can also be removed.

The best way to test and correct TECO programs is to write them in a fully structured style and test the structural units separately. It is a simple matter to split a large structured program into its units and insert the numeric values so that each execution path is tested once, which is sufficient. The individually tested units can then be recombined confidently.

Programmed TECO is not very different from ordinary interactive TECO; you have to keep a very clear picture of the pointer position. The new feature is the need to check the execution control logic. As stated before, you should take care to limit the use of logical operators to numerics which can *only* take the values Ø and (–1); this rule can save a lot of trouble.

You must expect infinite loops in programs. Be ready to strike CONTROL/C before the cost gets too high. Type-out messages in untested programs come into their own in this situation.

When an error in a program has been located the program must be copied into the buffer for editing. To do this you must be familiar with the commands described in chapter six. In general there are so many ESCAPE characters in programs that it is best to make all search and insertion commands @-modified. You must verify every change you make.

More on structural units

A strict pattern has been recommended for writing TECO programs. This is in the interest of clarity – one of the strong points in favour of structured programming in general. The structures which have been formally defined are the "if-then-else" and "repeat while/repeat until" units. However, while these may be sufficient for any programming task they are not always convenient. The additional structure of "case of" and "go to" do have their applications and can add clarity even though they involve unconditional transfer of execution across uncompleted levels of control.

The important thing is that the destination of the transfer is always a defined logical point such as the end of the case structure or to the end of the loop. What does not occur in structured programs, and what you must therefore take care to avoid in writing structured TECO, is unconditional transfer to some arbitrary destination.

The if-then structure

The simple "if-then" structure is quite common. This is really an "if-then-else" with the "else" part being null. This could be programmed in the manner shown previously with the comment !NULL! included after the !ELSE N! but there is no advantage in this and the straightforward approach is to use the following form:

```
!IF n"x NOT satisfied!    n"x OFIN N($)'
!THEN!
        (commands executed if n"x not satisfied)
!FIN N!
```

The command sequence "x (commands)' is itself an "if-then" structure. Provided that the command sequence to be executed if the test is satisfied is short there can be no objection to writing it directly. The considerations governing this are similar to those which govern the use of angle bracket structures. As a "rule of thumb" half-a-dozen commands is probably close to the limit and the "x and the ' should remain on the same line. For example, if the character after the pointer is a "B" it is desired to set a flag and pass over both the B and the following character, otherwise advance one character only. This can be programmed as follows:

```
0U1
!IF next=B set flag and skip!  (ϕA-↑↑B)"E - 1U1 C'
!skip character! C
```

It should be noted that in this example the test command is "E. If the structure had been programmed in the alternative form the logically complementary test "N would have been used to ensure the familiar "if-then" flow.

Case structure

It is usual in text editing programs for the course of command execution to be determined by the text itself. For example, if the character after the pointer is "A" a particular action may be taken, if it is "B" another, "C" yet another and so on. This can be programmed as a nested set of "if-then-else" structures but this is not always convenient and is often hard to follow. In such situations the "case" structure is often preferable. It is implemented as follows:

```
!CASE N!
        n"x  OCASE  N#1($)'
        m"x  OCASE  N#2($)'
        p"x  OCASE  N#3($)'
!other cases!
        (commands executed if none of n"x, m"x, p"x satisfied);
        OFIN N($)
!CASE N#1!
        (commands executed if n"x is satisfied)
        OFIN N($)
!CASE N#2!
        (commands executed if m"x is satisfied)
        OFIN N($)
!CASE N#3!
        (commands executed if p"x is satisfied)
!FIN N!
```

CASE N#M is read as "CASE structure N, number M".

In practice some care is required in applying the "case" structure in TECO. In particular the commands executed in each case must be complete units with one point of entry and one point of exit; in TECO it is often tempting to transfer out of one case directly into another. This may occur when the desired action depends on the next character or, in some cases, the next two characters in the text. This should be avoided. The correct procedure in such a case is use a series of case structures to analyze the text and generate the numerics (n,m,p above) which are then used to control execution of the final case structure which contains the commands to perform the desired actions. An example is given later.

The case structure can be expressed in a simplified form if the number of commands to be executed in a particular case is small. For example:

```
!CASE N!
!CASE N#1!     n"x  (commands) OFIN N $ '
!CASE N#2!     m"x (commands) OFIN N $ '
               p"x OCASE N 3 $ '
!other cases!  (commands)
          OFIN N $
!CASE N#3!     etc.
```

Unconditional transfer

Unconditional transfer should be to a logical point in the program but since it is not to the obvious end of a structure the destination should be made to stand out visually. One way of doing this is to enclose the tag within asterisks as shown below:

```
!GO TO!  0***TAG*** $
          .
          .
          .
!***TAG***!
```

The tag should have some significance itself. It might be
!***ERROR***! , for example.

One of the comparatively few cases where this command is genuinely useful is where a terminating condition is encountered deep within the program structure. In such a case it makes more sense to transfer directly to the end of the program than to return level by level carrying a termination "flag". Note that in this example transfer is indeed to a logical point in the program.

Repeat-break structure

Another command structure which is occasionally useful is the middle-tested loop or "repeat-break" structure. This can be implemented on TECO as follows:

```
!REPEAT N!
          (commands executed at least once and until
           n"x is satisfied)
!BREAK if n"x IS satisfied!    n"x OFIN N $ '
          (commands executed while n"x is NOT satisfied)
          OREPEAT N $
!FIN N!
```

This could be regarded as the fundamental "repeat" structure with the "repeat-while" and "repeat-until" structures as variants of it. However, it is conventional to give preference to the latter two structures where possible.

Formatting TECO programs

Paragraphing or indenting of the commands within a structural unit is an accepted and recommended formatting style in structured programming. The commands executed within a structure (this may include other structural units) are indented so that the limits of the structure (for example, !REPEAT N! and !FIN N!) stand out clearly.

The usual way to do this to to use the TAB facility. Unfortunately TECO uses TAB as a command. The solution is to use SPACE characters. This is not as inconvenient as may first appear. One way to use SPACEs is to type them in directly (an indentation of four SPACEs is sufficient) but this is not much fun if you have, say, five levels of indentation. The easiest procedure is avoid using TAB as a TECO command (no trouble) and enter the program using TAB characters for indenting. Then copy the program into the buffer and use the program

given below to change the TABs into SPACEs. This program interprets a TAB as a producing four-SPACE indentation rather than eight. Note that you can't use a program written with TABs until they have been changed. You must ensure that TABs which are wanted in search and insertion commands are *not* changed.

If you are entering the formatting program given below you could write it using TABs for indenting and with the command S<tab> (Ⓢ) replaced with $* (Ⓢ). Then use an FS command to change every TAB to four SPACEs and finally replace the * with TAB. The formatting program is more general than this; it correctly changes TABs even if they are not at the start of a line.

If your program has so many nested structural units that the necessary indentation is leaving you with very short lines you should consider whether or not it might be better all round to write the deeply indented sections as a separate macro. If the problem is simply one of space when entering the program with TABs you can start indenting this section afresh and edit in the necessary TABs to bring it into line with the rest of the program before changing the TABs to SPACEs.

One other problem introduced by the use of SPACEs for formatting concerns extending numeric expressions over more than one line. This may well introduce SPACEs into the expression which is disastrous in TECO-1Ø. The solution is to terminate the unfinished expression at the end of a line with a command Ui to store the partial value and then to commence the continuation of the expression on the next line (after any SPACEs) with the command Qi.

In case you are doubtful of the benefits of writing structured, commented TECO programs, see how long it takes you to work out what the following program does. This unstructured, uncommented example is taken from the TECO-1Ø Reference Manual:

!L! 0A-9"N!M! 0A-58"NCOM Ⓢ ' CD I<tab> Ⓢ ' LOL Ⓢ

You should also confirm that it ends with an error.

As an exercise, you could re-write the program in a formally structured style. You will probably find it hard to avoid making it end correctly!

The question of cost

The cost of using TECO for programmed editing cannot be ignored. Given that the file to be changed is worth editing in the first place, the cost of editing is influenced by the time taken to write and verify the program and the actual running cost. The emphasis in this book has been on effective techniques for writing programs but this must be seen in perspective.

For example, it is certainly possible to write a TECO program along the formal lines described to perform exactly the same function as
J<FNX Ⓢ Y Ⓢ ;>
following the detailed algorithm shown previously. It would, however, be inappropriate; wherever it is clear to do so use the standard high level TECO command forms. If you can think up the TECO command in your head (correctly!) it is safe enough to use it.

On the other hand there is nothing inherently wrong with using the formal structure. It is just that in *this* case the TECO program will take a little longer to write. This, however, brings us to the *main point* of TECO programming. Anything you can do with programmed TECO can be done with interactive TECO (in which you make the decisions on what is to be changed). You write a TECO program to save money (and time is money).

EXAMPLE 1

```
!This is a program to generate a software tab.  It is assumed
that there are no backspaces and that control characters
are echoed in a two character form except for escape and the carriage
control characters.!
[0 [1 [2 [3 [4 [5 [6
!The particular tabbing desired is stored in q-register 0;
the default is 4 column tabbing!
4U0
J 0U1 0U3 -1U4 0U6
!REPEAT 1!
!WHILE tabs remain!  :S $"E OFIN 1$'
    !Mark tab position!  -C .U5
    !IF it follows last insertion!  (.-Q1)"N 0ELSE 2$'
    !THEN insert spaces instead of the tab!
        D Q0<I $> .U1 (Q6+4)U6
        OFIN 2$
    !ELSE 2!
        0U2 (Q6+Q3)U6 0U3 0L
        !REPEAT 3!
        !WHILE start of line not found!  Q2"L OFIN 3$'
            !CASE 4!
            !CASE 4$1 at start!  (.-2)"L -1U2 OFIN 4$'
            !CASE 4$2 after carriage return!
                -2C (0A-13)"E 2C -1U2 OFIN 4$' 2C
            !Other cases back one line, count end of lines!
                %3$ -L
            !FIN 4!
            0REPEAT 3$
        !FIN 3!
        !IF on same line as last insertion!  (.-Q4)"N 0ELSE 5$'
        !THEN count carriage positions!  ((Q6-Q3)+(Q5-Q1))U6
            OFIN 5$
        !ELSE 5! !Mark start of line and count carriage positions!
            .U4 (Q5-Q4-Q3)U6
        !FIN 5!
        !Search for two-character echoing characters (not
        examined before)!
        Q1J
        !REPEAT 6!
        !Pass over strings of spaces!
            0A-^^ "E S^ES$ (.-Q5)"G Q5J''
        !BREAK when reach tab!  (.-Q5)"E OFIN 6$'
            !CASE 7!
            !CASE 7$1 null - control s!
                0A+8/8-1"E %6 OFIN 7$'
            !CASE 7$2 control n - control z!
                0A-1/13-1"E %6 OFIN 7$'
            !CASE 7$3 control characters codes 28 - 31!
                0A-24/4-1"E %6 OFIN 7$'
            !FIN 7!
            C
            0REPEAT 6$
        !FIN 6!
        !Insert spaces in place of tab, assuming no backspaces!
        Q5J D (Q6/Q0*Q0+Q0-Q6)U5 Q5<I $> .U1 (Q6+Q5)U6
    !FIN 2!
    0REPEAT 1$
!FIN 1!
]6 ]5 ]4 ]3 ]2 ]1 ]0
!End of program!
```

The most appropriate scheme of TECO programming is the one which is cheapest. In many cases this is the procedure which allows you to design, write and verify a program in the minimum time. The structured approach described here is extremely effective in meeting this requirement. The importance of program verification cannot be overemphasized; the damage that an incorrect TECO program can do to a file will be clear to you by now. Again, the structured approach may be justified on the score of easy logic verification alone.

The cost of writing and verifying a program is a once only cost. The other component of total cost is running cost and this depends on how many times the program is to be used. Running cost is made up of computer time and a charge for use of the computer core; the balance between these depends on the policy of the owner of the system. It is important to find out what TECO costs on your system.

The TECO program operates by interpreting each character in the command string and then taking the required execution action. Even if the command is in a loop and has been executed many times before it is examined each time it is encountered. The time taken to interpret the commands is quite significant and every character in the string is examined, even if it is ignored (as are comments). The time taken to execute the commands is often quite small in comparison. It is recommended that you find out how much it costs in your system to interpret commands. For example you could create a macro containing a 1ØØ character comment and execute it 1ØØ times to find out how much it costs to process about 1Ø ØØØ characters in a command string.

The other major cost in TECO is that associated with core use. To examine this you could place 1ØØØ characters in a Q-register and see how much it costs to copy it into the text buffer 1Ø times so that the buffer contains 1Ø ØØØ characters. Don't forget to clear the buffer afterwards. Continuing your investigation you could make a 1Ø page file with 4ØØØ characters per page give an EB command on this file followed by an EC (or EX) and see how much it costs to input and output that amount of text. You may be surprised.

To give a specific example, 25 ØØØ command characters were interpreted for the same cost as inputting and outputting 4ØØØ characters of text (about one full conventional page). The cost was about the same as 3Ø seconds of a programmer's time. The cost of interpretation increased by about 1Ø percent for each page stored in the buffers.

Clearly it is cheaper to run a program with a smaller number of characters in its commands than another which is equivalent but has a larger number of characters. If a program repeats a sequence of 5Ø command characters for *every* line of text the processing cost will be (using the costs quoted above) about one-eighth of the cost of input and output. If the number of command characters per line is halved it will only reduce the total cost by about five percent.

This saving may be worthwhile but should be judged against the cost of achieving it. If it is important, bear it in mind that efforts should be directed first at the most frequently executed sections of the program. If it is imperative to reduce the running cost you should seriously consider whether you should be using TECO.

The reason TECO was written in the first place is that by providing a large number of commands which are represented by a few letters but which perform complex operations it minimises the user's time in achieving editing tasks. The penalty is that the computer's time on the task is increased over what might be achieved by a dedicated user using a lower level language.

DECsystem-1Ø users might investigate XTEC, an experimental version of TECO-1Ø with the same commands but which compiles the command string once instead of interpreting each command as it is encountered. XTEC was written by Mark R. Crispin and is available from the Digital Equipment Users' Society (DECUS). It can give great savings on *some* programs but some features need to be investigated carefully (for example, passing numeric arguments to macros requires care).

It is, of course, quite possible to make the same changes to a file in different ways by programs which are *not* equivalent. It is possible in such cases to get very different running costs. It is the user's responsibility to select the best algorithm but structured TECO, by preserving

the algorithmic form, assists the user in this regard, especially when program modifications are being made.

This discussion leaves us with a problem. If a TECO program is to be understandable it should be fully commented. However an extensively commented program is somewhat more expensive to run than the same program without comments; how much more depends very much on the editing task.

The recommended solution *if* this problem is significant is to make two programs. One program is th source program. This should be fully commented and formatted; it is the one you write first, verify and retain for later modifications. The other is a stripped-down "run-only" version with all comments, other than those which double as tags for transfer commands, deleted, all non-text-argument SPACEs removed and key words such as REPEAT, ELSE, FIN and CASE abbreviated to R, E, F and C respectively. If the source program is formally written in the recommended style this can be done quite easily using interactive TECO. However this can also be achieved with a TECO program which may be easier if the source program has been written by someone who has not followed the advice given here.

Such a program is given below; it provides a good example of what can be done with TECO. On the first pass through the buffer unnecessary SPACEs are deleted and the tags referenced by O commands are marked with an asterisk and simplified if they contain key words. The corresponding O commands are simplified if necessary and additional occurrences of the referenced tag are deleted. On the next pass all comments except those starting with asterisks are deleted. Blank lines are also removed. During the passes commands are examined in detail and text arguments are skipped so that they are not altered in any way. The program is designed to be general; if it could have been assumed that exclamation marks did not occur inside text arguments the second pass could have been greatly simplified.

Since a large number of commands are executed per character in the buffer this program must itself be stripped down. This is best done by hand in the first instance.

This stripping-down process is sufficient to make considerable savings in some TECO-1Ø programs. However, this is not necessarily so with TECO-11. In detail, the operation of the unconditional transfer command in TECO-11 is rather different from that in TECO-1Ø. It will be recalled that TECO searches the current macro level for the first occurrence of the tag referenced in the O command. In TECO-11 this involves the detailed processing of every character in the macro level until the tag is found. For example, in the following section of a program:

```
!REPEAT 1!
    (a great number of characters)
    !REPEAT 2!
        (a few commands)
        OREPEAT 2 (Ⓢ)
        etc. (both loops end immediately)
```

each pass through the inner loop takes almost as long to process as one through the outer loop. The simple solution (provided a structured program has been written) is to store such inner structures as separate macros and to replace them in the main program with an Mi command. The depth of nesting of macros is system dependent but shouldn't be a problem. The sample program has been re-arranged in this way and the stripped-down version is up to ten times faster than a fully-commented version written as a single macro (TECO-11). The commented program remains the definitive version.

Concluding Remarks

You have now reached the end of the book and I hope you will apply TECO fruitfully in your own work. Don't get carried away though; TECO is only a means to an end!

EXAMPLE 2

```
!This is a program to remove all comments from a program unless
they are tags for G commands or start with an asterisk.  Tags
which are key words:REPEAT, ELSE, CASE and FIN are shortened to
R, E, C and F respectively.  Blank lines are also deleted.
text arguments are not altered in any way.!

!The program operates by skipping over text arguments, except for
G commands on the first pass, and passing commands which might end
with the letter G  to locate G commands.  The tag referenced by an G
command is found and marked with an asterisk and subsequent
occurrences of this tag are deleted.!

!The program operates in two passes through the buffer.  The first
is to mark the referenced tags; the second is to delete the
unnecessary comments.!

!The program will work under TECO-10, TECO-11 and XTEC
but it is not designed to strip XTEC programs (this is not necessary).!

[0 [1 [2 [3 [4 [5 [6 [7 [8 [Z [Y [X
!Set for exact searches to find tags!
^XU8 -1^X
!Set pass counter zero!
0U1
!Generate a macro to handle lower case commands!
!Any lower case letter is treated as an upper case one!
!This could be handled in TECO-11 with the ^u command!
!This particular form is used because "w and"v are
different in the two versions!
J@I\ 0AU0 (Q0-71)/26-1"E Q0-32U0'\
(0,,)XZ (0,,)K

!(N.B. Any comments inside a generated macro such as this will not be
deleted by this program since they occur inside a text argument)!

!Insert null to avoid going off page!
0I$
!REPEAT 1!
    !Initialize start of line marker!
    0U7
    !Commence two passes through the buffer!
    !REPEAT 2!
    !WHILE end of buffer not reached! (Z-,-1)"L 0FIN 2$'
        !Initialize flags!
        0U2
        !CASE 3 examine next character (in rough order of frequency)!
        !CASEs are:
                <space> - delete
                <blank lines> - delete
                <exclamation> - start of comment found
                S - start of single character text argument command
                I                                "
                N                                "
                _                                "
                ^I                               "
                ^R                               "
                F - start of two-character text argument command
                ^U (21)                          "
                ^ - check to see if next character makes command
                E                                "
```

```
        ^A - start of messase command
        O - start of "so to " command
        "<character> - two-character command; skip
        Q     "                       "
        U     "                       "
        %     "                       "
        [     "                       "
        ]     "                       "
        X     "                       "
        M     "                       "
        G     "                       "
        ^^    "                       "
    other cases: sinsle character command; skip!
    !Get the ascii value of the next character!
    MZ
    !Delete spaces on first pass!
    !N.B. Spaces often occur in strinss - caution:
    this will delete the tab in a space-tab combination!
    !It is a peculiarity of TECO-11 that the last space
    must be deleted separately!
    Q0-^^ "E .U4 :S^ES$"L (Q4,.)K' Q4J 0A-^^ "E D' OFIN 3$'

    !Delete blank lines!
    !A start of line marker is placed between the
    carriase return and the line feed !
    Q0-13"E C (0A-10)"E (.-Q7-2)"E -2D' .U7' OFIN 3$'

    !Generate comment case!
    Q0-^^!"E 1U2 OFIN 3$'

    !Skip two characters!
    Q0-^^""E 2C OFIN 3$'
    Q0-^^Q"E 2C OFIN 3$'
    Q0-^^U"E 2C OFIN 3$'
    Q0-^^%"E 2C OFIN 3$'
    Q0-^^^^"E 2C OFIN 3$'

    !Generate more cases!
    Q0-^^S"E 2U2 OFIN 3$'
    Q0-^^I"E 2U2 OFIN 3$'
    Q0-^^F"E 3U2 OFIN 3$'
    Q0-^^O"E 4U2 OFIN 3$'
    Q0-^^^A"E 5U2 OFIN 3$'
    Q0-^^^"E 6U2 OFIN 3$'
    Q0-^^E"E 7U2 OFIN 3$'

    !Skip two characters!
    Q0-^^["E 2C OFIN 3$'
    Q0-^^]"E 2C OFIN 3$'
    Q0-^^X"E 2C OFIN 3$'
    Q0-^^M"E 2C OFIN 3$'
    Q0-^^G"E 2C OFIN 3$'

    !Generate more cases!
    Q0-^^N"E 2U2 OFIN 3$'
    Q0-^^_"E 2U2 OFIN 3$'
    Q0-^^       "E 2U2 OFIN 3$'
    Q0-^^^R"E 2U2 OFIN 3$'
    Q0-21"E 3U2 OFIN 3$'
!Other cases,skip!
```

```
    C
!FIN 3!


!CASE 4!
    Q2-6"E OCASE 4#1$'
    Q2-7"E OCASE 4#2$'
    OFIN 4$
!CASE 4#1!
    !Caret found, check if next character makes command!

    C MZ
    !CASE 5!
    !CASEs are: ^A, ^I, ^R, ^U AND ^^!
        Q0-^^A"E -C 5U2 OFIN 5$'
        Q0-^^I"E -C 3U2 OFIN 5$'
        Q0-^^R"E -C 3U2 OFIN 5$'
        Q0-^^U"E -C 3U2 OFIN 5$'
        !Skip past ^^x!
        Q0-^^^"E 2C 0U2 OFIN 5$'
    !Otherwise skip both!
        C
    !FIN 5!
    OFIN 4$
!CASE 4#2!
    !E found, check if next character makes
    text command!
    C MZ
    !CASE 6!
    !CASEs are: EW, ER, EB, EG, EN AND EI!
    Q0-^^W"E -C 3U2 OFIN 6$'
    Q0-^^R"E -C 3U2 OFIN 6$'
    Q0-^^B"E -C 3U2 OFIN 6$'
    Q0-^^G"E -C 3U2 OFIN 6$'
    Q0-^^N"E -C 3U2 OFIN 6$'
    Q0-^^I"E -C 3U2 OFIN 6$'
    !Otherwise skip both!
    C
    !FIN 6!
!FIN 4!
!CASE 7!
    Q2-1"E OCASE 7#1$'
    Q2"N OCASE 7#2$'
!Other cases (I.E. Q2 = 0)!
    OFIN 7$
!CASE 7#1!
    !Start of comment found!
    !IF first pass skip! Q1"E 2S!$ OFIN 8$'
    !ELSE delete except if first character
        is * (delete  marker *)!
        0U3 .U4 C
        (0A-^^*)"E D -1U3'
        S!$
        Q3"E (Q4,,)K'
    !FIN 8!
    OFIN 7$
!CASE 7#2!
    !Text argument command found!
    !See if @-modified!
    -C
```

```
!IF @-modified! (0A-^^@)"N 0ELSE 9$'
!THEN move to text delimiter!
    (Q2-3)"E C' 2C
    !Store delimiter and senerate
    search macro to find second occurrence of delimiter!
    !N.B. Part of the text is copied in makins this macro!
    0AU3 .U4
    I2S$ Q3I$ @I/$/
    (Q4,.)XY (Q4,.)K MY
    !Record start of text!
    (Q4+1)U3
    0FIN 9$
!ELSE 9!
    !Move to start of text and record!
    (Q2-3)"E C' 2C .U3
    !IF not ^A command then skip to $!
        (Q2-5)"N @S/$/ 0FIN 10$'
    !ELSE senerate search macro for ^A!
        IS$ ^^^AI$ @I/$/
        (Q3,.)XY (Q3,.)K MY
    !FIN 10!
!FIN 9!
!Record end of text!
(.-1)U4
!CASE 11!
!CASE 11$1!!IF not 0 command! (Q2-4)"N 0FIN 11$'
!CASE 11$2!!IF not first pass! Q1"N 0FIN 11$'
!Other cases senerate search macro for up to
    first 35 characters of tas!
    !(necessary for TECO-10)!
    Q3J I:S!$ (Q4+3)J 0U0
    (Q4-Q3-35)"G (Q3+38)J -1U0'
    !N.B. If less than full tas is sousht
    it will not end with exclamation!
    @I/!$/ Q0"L -2C D C'
    (Q3,.)XY (Q3,Q3+3)K Q4J
    Q0"L Q3+35J' D Q0"E D'
    !N.B. Macro y is  :s<exclamation>tas<exclamation>$
    (no final <exclamation> if only part of the tas)!
    .U2 J
    !IF tas found! MY"E 0ELSE 12$'
        !Put * at start of tas, record lensth of
        and start of tas!
        (Q2-Q3)U5 -Q5C Q0+1"N -C %5' I*$  .U2
        !Simplify key words!
        !CASE 13!
        !FIN, ELSE, CASE and REPEAT become F, E, C and R!
            !N.B. Could use :S in TECO-11!
            (:SFIN$)$ (.-Q2-3)"E -2D 2U0 0FIN 13$'
            Q2J (:SELSE$)$ (.-Q2-4)"E -3D 3U0 0FIN 13$'
            Q2J (:SCASE$)$ (.-Q2-4)"E -3D 3U0 0FIN 13$'
            Q2J (:SREPEAT$)$ (.-Q2-6)"E -5D 5U0 0FIN 13$'
            0U0
        !FIN 13!
        !Return to callins 0 command and
        simplify it to asree with shortened tas!
        Q3J (Q2-Q3)"L (1-Q0)C' C Q0D
        !Allow for chanses to text!
        !(tas may come before or after 0 command)!
        (Q2-Q3)"L (1-Q0+Q4)U4' (Q4-Q0)U4
```

```
                    (Q2-Q7)"L (1-Q0+Q7)U7' J
                        !Delete extra occurrences of tag!
                        !(only the first occurrence is significant)!
                        !REPEAT 14!
                        !WHILE tag still found! MY+1"G OFIN 14$'
                            -(Q5+1)C .U2 2S!$ (Q2-.)U6 Q6D
                            !Allow for changes to text!
                            !(tag may come before or after o command)!
                            (Q2-Q3)"L (Q4+Q6)U4'
                            (Q2-Q7)"L (Q7+Q6)U7' J
                            OREPEAT 14$
                        !FIN 14!
                    OFIN 12$
                !ELSE 12!
                    !Look for *tag (may have been found before)!
                    !Generate a macro from macro y by adding
                    * after the :s!
                    J GY .U2 J 3C I*$
                    (0,Q2+1)XX (0,Q2+1)K
                    !IF not found! MX"L OFIN 15$'
                        !Look for *t (may have been simplified before)!
                        J GX .U2 J 4C
                        !Prepare to simplify O command!
                        !Look at first letter of tag (stored as macro x)!
                        !CASE 16!
                            0A-^^F"E 2U0 OFIN 16$'
                            0A-^^E"E 3U0 OFIN 16$'
                            0A-^^C"E 3U0 OFIN 16$'
                            0A-^^R"E 5U0 OFIN 16$'
                            0U0
                        !FIN 16!
                        !IF key word generate macro to search for
                        short tag E.G. :S<exclamation>F n<exclamation>$!
                            Q0"N C Q0D (0,Q2-Q0)XX'
                            (0,Q2-Q0)K
                        !IF not found! MX"L 0ELSE 17$'
                            GY ^A
    TAG ^A (2,,-1)T ^A NOT FOUND IN BUFFER.
^A
                            (0,.)K
                            OFIN 17$
                        !ELSE 17!
                            !Return to O command and simplify it!
                            Q3J C Q0D (Q4-Q0)U4
                        !FIN 17!
                    !FIN 15!
                !FIN 12!
            !FIN 11!
                !Move past end of text argument!
                (Q4+1)J
        !FIN 7!
        OREPEAT 2$
    !FIN 2!
    J C
!UNTIL 2 passes through buffer completed! %1-2"E OFIN 1$'
    OREPEAT 1$
!FIN 1!
!Delete null!
-D
!Restore search mode!
```

```
Q8^X
]X ]Y ]Z ]8 ]7 ]6 ]5 ]4 ]3 ]2 ]1 ]0
!End of program!
```

****

THE PROGRAM IS SHOWN HERE AS A SINGLE PROCEDURE
FOR CLARITY. IN PRACTICE IT IS USED AS A SET
OF MACROS.

****

This program was written on an upper case only terminal.
A TECO program was used to convert the comments to
lower case.

APPENDIX

CHAPTER-BY-CHAPTER SUMMARY OF TECO COMMANDS

COMMANDS EXPLAINED IN CHAPTER ONE

Commands which return a numeric value

B	Beginning of buffer; returns zero (∅).
Z	End of buffer; returns number of characters in buffer.
.	Pointer position; returns the number of characters to the left of the pointer.
H	Whole buffer; same as (B, Z) or (∅, Z).
EH	Returns current EH setting.
EV	TECO-11 only: Returns current EV setting.
(↑↑) x	Returns ASCII code of character x.

Pointer movement commands

nJ	Jump to position n; i.e. between n^{th} and $n + 1^{st}$ characters.
J	Jump to start of buffer; same as ∅J.
nC	Move pointer forward over n characters; same as (. + n)J.
nR	Move pointer back (Reverse) over n characters; same as –nC.
nL	Move pointer to position immediately after n^{th} following end-of-Line character.
L	Move to start of next "line"; same as 1L.
∅L	Move to start of current "line".
–L	Move to start of previous "line"; same as –1L.

Type-out commands

nT	Type out all characters between pointer and the n^{th} following end-of-line character; pointer not moved. Format as for nL.
(m, n)T	Type out $m + 1^{st}$ through n^{th} characters; $m < n$; pointer not moved.
n (↑T)	TECO-11 only: Type out the character whose (decimal) ASCII code is n.
nV	TECO-11 only: Verify; type out (n – 1) lines on either side of the pointer.
nEV	TECO-11 only: Enable verify; type before asterisk prompt; n = ∅; no effect (default setting) n = –1; type current line; n = 1 – 31; type current line with LINE FEED after pointer; n > 31; type current line with character whose code is n after the pointer; n = (m*256 + n); as above but also types (m – 1) lines either side of current line. no argument; return current value of n.
n=	n Equals; type out the value of the numeric argument n.
FORM FEED or (↑L)	When executed as a command causes a form feed action.

Deletion commands

nD	Delete n characters after pointer. Format same as for nC.
(m, n)D	TECO-11 only: same as (m, n)K
nK	Delete (Kill) all characters between the pointer and the n^{th} following end-of-line character. Format same as for nL.
(m, n)K	Delete (Kill) the m + 1^{st} through n^{th} characters. Format as for (m, n)T; automatically performs mJ command.

Insertion commands

I . . . text . . . (\$)	Insert the character string . . . text . . . , which is delimited by I and (\$), immediately to the left of the pointer. That is, the pointer is located at the end of the insertion.
<tab> . . . text . . . (\$)	Same as I <tab> . . . text . . . (\$); not recommended.

Error commands

?	As first command after an error message; types out part of the command string ending with the command which caused the error.
/	TECO-1Ø only: used like ? and can be typed before or after ?; types out an expanded description of the error.
nEH	Enable Help: n = Ø or n = 2; type error code and message (default setting) n = 1; type code only. no argument; return current value of n. TECO-1Ø only: n = 3; type code, message and expanded description of error. TECO-11 only: n = 3; type code, message and execute ? command.

Command string editing characters

RUBOUT	Deletes last character in command string
(↑U)	Deletes back to, but not including last <cr><lf> pair in command.
(↑G) SPACE	Types out correct version of command string from last <cr><lf> pair to the end.
(↑G) *	TECO-11 only: types out the whole of the current version of the command string.
(↑G) (↑G)	Aborts whole command (if given before execution has started).
(\$) (\$)	Command string complete; start execution.

Execution control commands

(\$)	Interpretation of preceding command completed.
(\$) (\$)	TECO-10: in command string - return to command level; not recommended.
(↑C)	TECO-1Ø: not a TECO command; execution not in progress - return to system; has other functions. TECO-11: as a command; return to system when preceding commands in string have been executed.
(↑Z)	TECO-1Ø: as a command; return to monitor when preceding commands have been executed.

Execution control commands - continued

n<...commands...>	Execute ...commands... thus: n > Ø; execute n times, n ≤ Ø; skip execution, no argument; repeat execution indefinitely.
^ or ↑	Interpret next character in command string as a control character.

Abort commands

(↑C)(↑C)	TECO-1Ø: command execution in progress; stop execution, return to monitor. TECO-11: command execution in progress; stop execution, accept new command string.
(↑O)	Command execution in progress: stop all type-out, continue execution.

System commands (see Reference Manuals)

.R TECO (DECsystem-1Ø and RT-11)	Enter TECO; all buffers empty;
RUN $TECO (RSTS/E)	all options (e.g. EH) have default values.
TECO (RSX-11)	
.REENTER (DECsystem -1Ø) and RT-11)	Re-enter TECO provided no other programs have overwritten core; text buffer and pointer not changed; command buffer reinitialized.
(↑C) (DECsystem -1Ø)	Return to monitor immediately.

Numeric expressions

+	Between terms: addition.
–	Between terms: subtraction; Before a term: negation.
SPACE	TECO-1Ø: same as +: TECO - 11: no significance; The use of space characters in or immediately after numeric expressions should be avoided.
(expression)	Indicates order of evaluation of terms of an expression; use freely.

Formatting of commands

Text arguments	All (allowable) characters in a text argument are significant.
<cr> and <lf>	Can be used freely outside text arguments.
SPACE (or <space>)	Can be used freely outside text arguments except where numeric arguments are involved.

COMMANDS EXPLAINED IN CHAPTER TWO

Commands which return a numeric value

ES	Returns the current ES setting.

The following apply to TECO-11 only:

m ↑Q	Returns number of characters between pointer and m^{th} following LINE FEED character.
:Qi	Returns the number of characters in the text in Q-register i.
↑S	Returns negative of length of last insertion or successful search string.
↑Y	Returns (m, n) arguments to cover last insert or successful search, provided pointer at end of string; same as (.+↑S,.).
↑Z	Returns the total number of characters stored in the Q- registers.

Type-out commands

The following apply to TECO-11 only:

:Gi	Type out text in Q-register i; does not affect buffers.
:G_ or :G←	Type out the most recent search string.

Search commands

nS...text...$	Searches for . . . text . . . starting at the pointer: n positive: searches for n^{th} occurrence of string after pointer; no argument: n = 1 is assumed; if successful pointer left at end of matched string; if unsuccessful pointer left at start of buffer and error occurs.

Note: a search failure inside angle brackets is not an error.

	TECO-11 only: n negative: searches towards start of buffer.
ns$	The last search command text argument string executed is assumed to apply.
nES	Enable automatic type-out after a successful search: n = ∅: no effect (default setting); n = –1: type line containing matched string; n = 1 – 31: type line with LINE FEED after pointer; n > 31: type line with character whose code is n after the pointer; n = (m*256 + n): as above but also types out (m – 1) lines on either side of line with matched string; no argument: returns current value of n.

TECO-11 only:

(m, n)S...text...$	Bounded search; n as for normal search but pointer not moved more than ABS(m) – 1 positions in obtaining n matches of string; if search is unsuccessful pointer is not moved; m = ∅: the (∅, n)S command is equivalent to an nS command with no pointer movement on failure.

Search commands – continued

(m, n)S($) — The last search command text argument string executed is assumed to apply.

Replacement commands

nFS...text 1...($)...text 2...($)

As for nS...text...($) command except that if search is successful the matched string, ...text 1... is deleted and replaced with ...text 2... .

The following apply to TECO-11 only:

(m, n)FS...text 1...($)...text 2...($)

As above but search is an (m, n) search.

(↑R)...text 1...($)...text 2...($)

Alternative form for FS command.

FR...text...($) — Deletes number of characters equal to length of the last insertion or successful search string to left of pointer and replaces them with ...text... ; an S command followed by an FR command is equivalent to an FS command.

Store and recall commands

Note: the letter i in these commands stands for any letter or digit (alphabetic case is not significant).

nXi — Copy all characters between pointer and the n^{th} following end-of-line character in Q-register i; pointer and buffer not affected. Q-register i overwritten.

(m, n)Xi — Copy m + 1^{st} through n^{th} characters in Q-register i; m > n; pointer and buffer not affected; Q-register i overwritten.

*i — As first command after (↑G) (↑G) command string abortion: copy contents of command string buffer in Q-register i; Q-register i overwritten.

Gi — Copy text in Q-register i into buffer to left of pointer; Q-register i not affected.

The following apply to TECO-11 only:

↑U...text...($) — Insert ...text... into Q-register i; Q-register i overwritten.

G_ or G← — Copy last search command text argument string into buffer to left of pointer;

Execution control command

; — After a search command in a command string enclosed in angle brackets: ignored if search is successful; if search fails execution passes to first command after corresponding closing angle bracket.

COMMANDS EXPLAINED IN CHAPTER THREE

Commands which return a numeric value

ED	Returns the current ED setting
↑E	FORM FEED flag: returns –1 if last input terminated by FORM FEED; otherwise Ø.
↑N	End of file flag: returns –1 if last input terminated by end of file; otherwise Ø.

Output commands

EF	Close current output file.
EK	TECO-11 only: remove all reference to current output file.
EWfilespec$	Open new file described by filespec to receive output.
(m, n)P	Copy m + 1st through nth characters in buffer into output file; no effect on buffer or pointer.
(m, n)PW	Alternative form for (m, n)P; not recommended.
nPW	Copy buffer into output file n times, appending a FORM FEED to the buffer contents each time.

Input commands

A	Append next page of input file to end of text in buffer; if input file not in pages input to end of file or to limit imposed by core available; no effect on pointer.
nED	TECO-11 only: default setting is system-dependent; has several effects but if: n = –1: Y works as described; n = Ø or 1: Y works as described if no output file open, otherwise Y command is invalid.
ERfilespec$	Pre-set existing file described by filespec for input.
Y	Clear the buffer before appending next page of input; equivalent to HKA; not recommended.

System commands (see Reference Manuals)

.MAKE filespec	(DECsystem-1Ø)
MAKE filespec	(RSTS/E and RSX-11)
.EDIT/CREATE filespec	(RT-11, TECO as default editor)

These commands are equivalent to the appropriate system command to enter TECO, followed by the TECO command:
EWfilespec$$.

COMMANDS EXPLAINED IN CHAPTER FOUR

Input/output commands

EBfilespec(\$)	Preset the specified file for input and open a new file with the same filespec for output.
EC	TECO-11 only: output current buffer and transfer rest of input file to output file.
EG(\$)	TECO-1Ø: perform EX command and re-execute last compile class system command; TECO-11 under RSX-11: same as EX.
EG . . . string . . .(\$)	TECO-11 under RT-11: perform EX command and execute . . . string . . . as system command; TECO-11 under RSTS/E: perform EX command and execute . . . string . . . as system command according to filename extension of last output file closed.
EX	Output current buffer, transfer rest of input file to output file and return to system (exit from TECO).
FN . . . text 1 . . .(\$) . . . text 2 . . .(\$)	Performs an N . . . text . . .(\$) search and if successful replaces the matched text with . . . text 2 . . . .
N . . . text . . .(\$)	Searches buffer for . . . text . . . like an S . . . text . . .(\$) command but if unsuccessful outputs the buffer and inputs the next page from the input file. Numeric arguments as for S command.
_ . . . text . . .(\$)	Like N . . . text . . .(\$) command except that when the buffer has been searched without success it is deleted before the next page is input.
nP	Next Page: outputs current buffer and clears it then inputs next page from the input file; this sequence performed n times.

Store and recall commands

The following apply to TECO-11 only:

G*	Copy filespec used in the last EB, EI, EN, ER or EW command from the filespec buffer into text buffer to left of pointer.
: G*	Type out the most recent filespec; does not affect buffers.

Secondary input/output stream commands

The following apply to TECO-11 only:

EA	Enable the secondary output stream.
EP	Enable the secondary input stream.
ER(\$)	Enable the primary input stream (default condition).
EW(\$)	Enable the primary output stream (default condition).

"Wild card" file specification commands

The following apply to TECO-11 only:

ENfilespec(\$)	Enables "wild card" filespec matching.
EN(\$)	Enter next filespec which matches pre-set "wild card" form into filespec buffer.

System commands (see Reference Manuals)

.TECO filespec	(DECsystem-10)
.EDIT filespec	(RT-11, TECO as default editor)
TECO filespec	(RSTS/E and RSX-11)

These commands are equivalent to the appropriate system command to enter TECO followed by the TECO command:

EBfilespec (ESC) HK A(ESC)(ESC)

.CLOSE (DECsystem-10 and RT-11) — Closes an output file left open on exit from TECO; not recommended except in emergency.

COMMANDS EXPLAINED IN CHAPTER FIVE

Storage and recall commands

Note: the letter i in these commands stands for any letter or digit (alphabetic case is not significant)

*i	As first command after (↑G) (↑G) command string abortion or after command execution: copy contents of command string buffer into Q-register i; Q-register i overwritten.
Mi	Copy text in Q-register i into command execution string; Q-register i not affected.
(↑C)	TECO-11: within macro - return to command level. TECO-1ϕ: not a command - has same effect as in TECO-11 but caused by error condition.

COMMANDS EXPLAINED IN CHAPTER SIX

Commands which return a numeric value

ED	Returns current ED setting.
ET	Returns current ET setting.
↑X (outside argument)	Returns current search mode setting.

Insertion command

nI $	Insert the single character whose (decimal) ASCII code is n to the left of the pointer.

Text specification commands

@(command)/...text.../	Text argument delimited by arbitrary character (e.g. /) after command and its next occurrence; delimiters do not appear in text; ESCAPE characters can be included in text.
	TECO-1Ø: applies to search and insertion text arguments only.
CONTROL characters	TECO-1Ø only: the control characters, except ↑C, ↑O, ↑U and ↑H – ↑M, are reserved inside search and insertion text arguments as commands (not all yet defined).

Search string matching commands

Note: The following commands mean that at the place in the search string occupied by the command the specified characters are to be accepted as a match.

↑E A	Any alphabetic character.
↑E D	Any digit.
↑E L	Any line terminator.
↑E S	Any string of SPACE or TAB characters.

The following apply to TECO-1Ø only:

↑E V	Any lower case alphabetic.
↑E W	Any upper case alphabetic.
↑E <nnn>	The character whose octal ASCII code is nnn.
↑E [a, b, c...]	Any one of the ASCII characters a, b, c.... .

The following apply to TECO-11 only:

↑E C	Any RAD5Ø character (alphanumeric, PERIOD or DOLLAR).
↑E R	Any alphanumeric.
↑E X	Same as ↑X.
↑E Qi	The characters which match the text stored in Q-register i.
↑E Q*	The characters which match the last filespec.
↑E Q_	The characters which match the last search text argument.

Search string matching commands - continued

The following apply to both TECO-1Ø and TECO-11:

(↑N) x	Any character except the specified character x.
(↑S)	Any separator character; TECO-11: Any character which is not an alphanumeric; TECO-1Ø: Any character which is not an alphanumeric, DOLLAR, PERCENT or PERIOD.
(↑X) (inside argument)	Any character; a character must be present at this position.

Text argument interpretation commands

nED	TECO-11 only: enables CARET command inside text arguments; n = –1 or Ø; CARET enabled; n = 1; CARET disabled; also effects Y command.
(↑Q) (inside argument)	Take next character literally; does not apply to characters with immediate control functions; TECO-1Ø: does not apply to ESCAPE.
(↑R) (inside argument)	TECO-1Ø only: like (↑Q) but applies to ESCAPE.
(↑T) (inside argument)	TECO-1Ø only: take all following characters in this text argument literally except (↑R) and (↑T) ; cancelled by another (↑T) .
↑ (CARET or UPARROW)	TECO-11 only: Take next character as CONTROL/character.

Terminal control commands

nET	TECO-1Ø: n = Ø; enable standard echoeing; n ≠ Ø; enable literal type-out. TECO-11 only: Bit Ø (1): enable literal type-out; Bit 1 (2): see Reference Manual; (V.D.U. terminals); Bit 2 (4): transmit lower case; Bit 4 (16): cancel (↑O) after type-out; also has other Bit functions.
nEU	Enables case flagging on type-out: n = –1: no case flagging; n = Ø: lower case characters flagged: a becomes 'A; n = 1: upper case characters flagged.

Alphabetic case control commands

n (↑X) (outside argument)	Search mode: n = Ø: either case mode; match achieved regardless of case; n = 1: exact case mode; match achieved only if alphabetic cases match.
(↑↑) x (inside argument)	TECO-11: character x taken to be lower case equivalent; TECO-1Ø: character x taken to be lower case equivalent but only applies to non-alphabetic characters.

Alphabetic case control commands

The following apply to TECO-1Ø only:

↑V (outside argument)	All characters to be taken as upper case unless overruled.
Ø ↑V (outside argument)	Restores default condition; no case conversion.
↑V ↑V (inside argument)	All following characters in this argument taken as lower case unless overruled; exact mode implied in search.
↑V (inside argument)	Take next character only in lower case; exact search implied.
↑W (outside argument)	All characters to be taken as upper case unless overruled.
Ø ↑W (outside argument)	same as Ø ↑V .
↑W ↑W (inside argument)	All following characters in this argument taken as upper case unless overruled; exact mode implied in search.
↑W (inside argument)	Take next character only in upper case; exact search implied.

COMMANDS EXPLAINED IN CHAPTER SEVEN

Input commands

The following apply to TECO-11 only:

EIfilespec (\$)	Enables indirect input; next TECO request for input from the terminal comes from referenced file; not available under RT-11.
EI (\$)	Closes indirect input file.

Execution control commands

'	Apostrophe: marker for conditional execution command; used in matching pairs with commands n"x; is a command terminator.
n"E	n equal to ∅: ignored; n not equal to ∅: execution passes to command after matching apostrophe.
n"G	n greater than ∅: ignored; n not greater than ∅: execution passes to command after matching apostrophe.
n"L	n less than ∅: ignored; n not less than ∅: execution passes to command after matching apostrophe.
n"N	n not equal to ∅: ignored; n equal to ∅: execution passes to command after matching apostrophe.
Otag (\$)	Unconditional transfer of execution to command following first appearance of !tag! in current macro level.
n;	Within angle brackets only: n negative: ignored; n = ∅ or positive: execution passes to command following corresponding closing angle bracket.

Comment or tag command

!...text...!	Ignored except as destination of an unconditional transfer of execution command.

System commands

The following apply to TECO-11 only:

.EDIT/EXECUTE [: text] filespec	RT-11: Enters first page of named file in Q-register Z, enters the text into the buffer and executes the text in Q-register Z as a macro.
MUNG filespec, text	RSTS/E and RSX-11: Inserts the text into the buffer and enables the named file for indirect entry in response to the request for terminal input.
TECO @ filespec	RSX-11: as above but does not insert text into the buffer.

COMMANDS EXPLAINED IN CHAPTER EIGHT

Numeric operator commands

There is no hierarchy of operators

*	Multiplication.
/	Integer division (remainder lost).
&	Bitwise logical AND.
#	Bitwise logical OR.
n (↑_)	TECO-11 only: equivalent to –(n + 1); used to form logical complement.
n: =	TECO-11 only: type-out n and leave carriage at end of type-out.

Octal and decimal conversion commands

(↑D)	TECO-11 only: accept all following numeric input as decimal; restores default condition.
↑ O	CARET-O: TECO-1Ø: accept following numeric input as octal; applies to next input only; TECO-11: accept all following numeric input as octal.
n==	Type out the decimal numeric n as an octal number.
n:==	Type out n as an octal number and leave carriage at end of type-out.

Encoding and Decoding commands

n\	Insert the ASCII character representation of n to the left of the pointer.
\	Return the numeric value of the string of ASCII digits to the right of the pointer.

Commands which return numeric values

nA	TECO-11: returns value of decimal ASCII code of $(.+n+1)^{th}$ character in buffer; ØA returns code of next character after pointer; TECO-1Ø: returns value of next character after pointer regardless of n; use ØA.
EO	TECO-1Ø only: returns EO value of TECO program.
(↑B)	TECO-11 only: returns current date (system-dependent).
(↑F)	Returns current value of the console switch register.
↑H	Returns time-of-day (system-dependent); use CARET-H form to avoid BACKSPACE.
(↑V)	TECO-11 only: returns version number of TECO program.

Q-register commands

Qi	Return the value of the numeric stored in Q-register i.
nUi	Store the numeric n in Q-register i; Q-register i overwritten.
%i	Add 1 to the numeric stored in Q-register i and return the incremented value.
n%i	TECO-11 only: add n to the numeric stored in Q-register i and return the incremented value.

Colon-modified commands

In TECO-1Ø this applies to search and insertion commands only.

:Command	Modifies commands which may succeed or fail depending on state of buffer etc. : if command succeeds returns the value (–1); if command fails returns the value Ø.
::S...text...(ESC)	Equivalent to (1, 1):S...text...(ESC) .

Execution control commands

nEO	Enables old version of TECO-1Ø: n = 1 versions up to 21; n = 2 versions up to 23B.
n"A	n is ASCII code of alphabetic: ignored; n is not ASCII code of alphabetic: execution passes to command after matching apostrophe.
n"C	n is ASCII code of alphanumeric, $ or . (TECO-1Ø: also %): ignored; n is not ASCII code of one of above: execution passes to command after matching apostrophe.
n"D	n is ASCII code of digit: ignore; n is not ASCII code of digit: execution passes to command after matching apostrophe.
n"F	Same as n"E.
n"R	TECO-11 only: n is ASCII code of an alphanumeric: ignore; n is not ASCII code of alphanumeric: execution passes to command after matching apostrophe.
n"S	Same as n"L.
n"T	Same as n"L.
n"U	Same as n"E.
n"V	TECO-1Ø: n is ASCII code of lower case alphabetic: ignored; n is not ASCII code or lower case alphabetic: execution passes to command after matching apostrophe. TECO-11. n is ASCII code of upper case alphabetic: ignored; n is not ASCII code of upper case alphabetic: execution passes to command after matching apostrophe.
n"W	TECO-1Ø: as for n"V in TECO-11. TECO-11: as for n"V in TECO-1Ø.

COMMANDS EXPLAINED IN CHAPTER NINE

Interactive programming commands

↑A ... message ... ↑A	Types out ... message ... : the second A must be a CONTROL/A; TECO-11 only: the @-modified form may be used.
↑T	Execution pauses until one character has been entered through the terminal; the ASCII code of this character is returned as a numeric value.

Q-register commands

In TECO-11 only a numeric argument can pass across the following commands.

[i	Copy the contents of Q-register i to the top of the Pushdown List; List cleared when execution terminates.
]i	Copy contents of top register of Pushdown List into Q-register i; Q-register i overwritten.

Execution trace command

?	Not first command after error: causes TECO to type out each following command as it is executed; cancelled by another ?; TECO-11 only: cancelled at end of current execution.

Terminal control commands

ET	TECO-11 only: Bit 5 (32): read with no wait; ↑T command returns value (–1) with no pause in execution if no character has been typed already; Bit 6 (64): detach terminal; Bit 7 (128): terminate TECO if error occurs. Bit 15 (–32768): switched off if CONTROL/C is typed.