intel®

EDIT REFERENCE MANUAL

Copyright © 1981, Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 143587-001

EDIT REFERENCE MANUAL

Order Number: 143587-001

Copyright © 1981 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	PRINT DATE
-001	Original Issue	8/81

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BXP	Intelevision	Micromap
CREDIT	Intellec	Multibus
i	iRMX	Multimodule
ICE	iSBC	Plug-A-Bubble
iCS	iSBX	PROMPT
im	Library Manager	Promware
INSITE	MCS	RMX/80
Intel	Megachassis	System 2000
Int$_e$l	Micromainframe	UPI
		μScope

and the combination of ICE, iCS, iRMX, iSBC, iSEX, MCS, or RMX and a numerical suffix.

PREFACE

This manual documents EDIT, an iRMX 86-based text editor which allows you to create and modify files. It contains introductory and tutorial material as well as detailed descriptions of all the EDIT commands.

Reader Level

This manual is intended for both the inexperienced and experienced EDIT user. The unfamiliar user may begin at the introduction, continue through the tutorial, and finally learn advanced editing techniques.

On the other hand, the informed user can use the Command Dictionary and the alphabetically tabbed command chapter (Chapter 5) for quick reference.

Conventions

This manual contains examples of EDIT commands entered at the terminal. In the examples, the lines that you are supposed to enter are printed in **boldface** type. This distinguishes your input from EDIT's output. This manual also indicates an end-of-line with a (c/r) and tab spacing with a (T) for the first few examples. Thereafter, the carriage return and tabs are not explicitly shown.

Related Publication

The following manual provides additional background and reference information.

- *iRMX 86™ Human Interface Reference Manual,* 9803202

CONTENTS

CONTENTS (Continued)

TABLES

CHAPTER 1
INTRODUCTION

What Is An Editor?

An editor is a program which allows you to create and modify files. All interactive computing systems have some form of editing facility; however, the features and editing capabilities can vary greatly.

A typical line editor consists of a set of commands which you can use both for quick changes and for more complex editing functions. Line editors are generally streamlined products. They display only the text for which you specifically ask. Because a line editor is not dependent upon a screen, it is useful both in systems which have screens and in systems which have a hard-copy terminal such as a teletypewriter.

What Is EDIT?

EDIT is an interactive line editor. It is not only streamlined but also very powerful. You can use it to create and modify text that can be anything from a document to a sophisticated program. The best way to learn EDIT is to read this manual while using EDIT to do the examples. You should also read the descriptions referenced in the Command Dictionary because not all the commands are included in the Tutorial.

If you are already familiar with EDIT's basic commands you may wish to skip the Tutorial and use the Command Dictionary exclusively. This dictionary tells you where to find the pictorial syntax and a detailed description of the desired command. This manual also includes chapters on Special Characters and Advanced Editing that describe more sophisticated editing techniques.

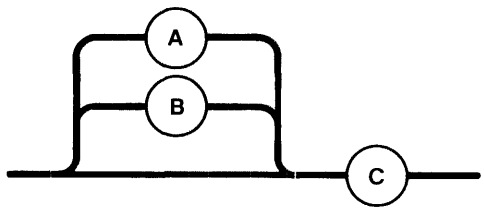
How Does EDIT Work?

The EDIT program saves your text on temporary, internal storage called a buffer. This buffer allows you to create and change text before you put it into more permanent storage. If you want to edit a file that is already in secondary storage, EDIT copies the contents of that file into the buffer. You can then begin to alter the text. However, the content of the file on secondary storage does not change until you execute one of the EDIT commands specifically designed to write the text in the buffer onto secondary storage.

EDIT also maintains a marker which points to something called the "current line." EDIT keeps track of the line that you are editing (the current line) by moving the marker in response to the commands you type. The marker remains at this position until another command causes it to move. Some commands cause the marker to move whereas others do not. Chapter 5 describes the marker position for individual commands.

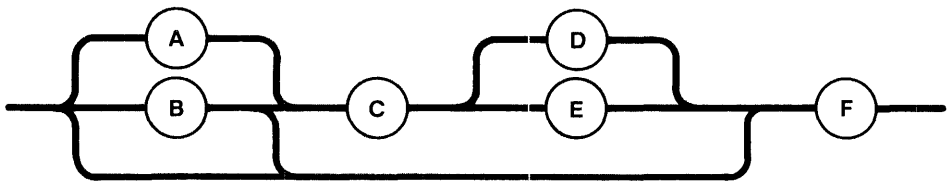
Pictorial Representation of Syntax

This manual uses a schematic device to illustrate the syntax of commands. The schematic consists of what looks like an arerial view of a model railroad setup, with syntactic entities scattered along the track. Imagine that a train enters the system at the upper left, drives around as much as it can or wants to (sharp turns and backing up are not allowed), and finally departs at the lower right. The command it generates in so doing consists of the syntactic entities that it encounters on its journey. The following pictorial syntax shows two ways (A or B) of reaching "C."



x-116

The schematics do get more complicated, but just remember that you can begin at any point on the left side of the track and take any route to get to the end as long as you do not back up. Trace the following track and note that there are many different combinations possible.



x-117

CHAPTER 2
TUTORIAL

This tutorial is designed to illustrate most of the EDIT commands by having you create and alter text. The text you will work with is a PL/M-86 program. This program will make change for a dollar in the form of half dollars, quarters, dimes, nickels, and pennies. The text enclosed in the symbols "/*----*/" is a comment in the PL/M language and is ignored by the PL/M-86 compiler. These comments are useful only as an aid in the reading of the program. However, you do *not* have to understand PL/M-86 if you treat the program simply as a document or as lines of prose. If you do know the language you may catch some mistakes as you type the following program. Do *not* correct the program as you type. The mistakes are intentional and you will correct them later with the EDIT commands.

Invoking EDIT

Before you can create any text you must first invoke the EDIT program. The simplest way to enter EDIT is to type:

```
ED (c/r)
```

EDIT responds by identifying itself and giving you an asterisk (*) as a prompt. An example of a sign-on message is as follows.

```
iRMX 86 Line Editor, V1.0
*
```

You can enter commands in upper- or lower-case but to avoid confusion this manual will use upper-case.

Error Message

If you type anything that EDIT doesn't understand, it will respond with a question mark (?). This is the only error message in the EDIT program. So whether you have made a syntactical error or a typographical error, EDIT will respond with a question mark.

General Form

A general form for a command in this manual consists of hyphenated words and a capitalized command letter. When you enter the command, you will substitute actual line numbers or symbols for the hyphenated words. In contrast, the command letter appears exactly as it should be entered.

Creating The Text — Appending

In order to create some text you will have to use the Append command (A). Append the following program as shown. You may use either tabs or blanks to obtain indentation.

```
*A (c/r)
make$change: (c/r)
(T) DO;
(T)  (T)  DECLARE money(8) BYTE;  (T)  (T)        /*this is the result*/
          DECLARE change BYTE;                    /*number to be converted*/
          DECLARE change BYTE;                    /*number to be converted*/

          next$money:  (T)  (T)  (T)  (T)         /*this is a procedure*/
(T)  (T)  (T)  (T)  DECLARE X BYTE;               /*X is specified */
                    money(I) = X;
(T)  (T)  (T)  PROCEDURE(X);
                    I = I+1;
                    change = change - X;
               END
next$money;

          change = 100 -    ;                     /*wriye the cost here*/
          I = 0;                                  /*initialize the index index*/
          IF change >= 50 THEN                    /*half dollar*/
               CALL next$money(50);
          END;
          DO WHILE change >= 25;                  /*quarters*/
               CALL next$money(25);
          DONE WHILE change >= 10;                /*dimes*/
               CALL next$money(10);
          END;
          DO WHILE change >= 15;                  /*nickels*/
               CALL next$money(5)
          END;
          DO WHILE change >= 1;                   /*pennies*/
               CALL next$money(1);
          END;
          DO WHILE change >= 8;
               CALL next$money(0)                 /*zero out rest of coins*/
          END;
     END make$change;
.
*
```

In order to stop appending you must type a period (.) followed by a carriage return on an otherwise blank line.

Adding Line Numbers

Now you can put line numbers on the buffer for easy reference during the rest of the EDIT sesion. To do this, type:

```
*N
```

EDIT numbers the lines but only displays the prompt.

```
*
```

Do not type another "N" command at this time. The Numbering command (N) can not only turn on the numbers but it can also turn them off. If you ever want to get rid of the numbers associated with your text, you can type the Numbering command (N) again and EDIT will delete them. These numbers are displayed with your text, but they are not actually part of the file.

Printing

To verify that the Numbering command was executed, you can use the Print command (P). The general form for Print (P) is:

```
starting-line-number, ending-line-number P
```

You already know the text started on line one but you might not be sure at which number it ended. You can use the dollar sign ($) to refer to the last line of the buffer.

```
*1,$P                    (starting-line-number = 1,  ending-line-number = $)
```

The EDIT program responds by displaying the contents of the buffer followed by a prompt.

```
1:  make$change:
2:    DO;
3:       DECLARE money(8) BYTE;                  /*this is the result*/
4:       DECLARE change BYTE;                    /*number to be converted*/
5:       DECLARE change BYTE;                    /*number to be converted*/
6:
7:       next$money:                             /*this is a procedure*/
8:              DECLARE X BYTE;                  /*X is specified */
9:              money(I) = X;
10:          PROCEDURE(X);
11:             I = I+1;
12:             change = change - X;
13:          END
14: next$money;
15:
16:      change = 100 -    ;                     /*wriye the cost here*/
17:      I = 0;                                  /*initialize the index index*/
18:      IF change >= 50 THEN                    /*half dollar*/
19:          CALL next$money(50);
20:      END;
21:      DO WHILE change >= 25;                  /*quarters*/
22:          CALL next$money(25);
23:      DONE WHILE change >= 10;                /*dimes*/
24:          CALL next$money(10);
25:      END;
26:      DO WHILE change >= 15;                  /*nickels*/
27:          CALL next$money(5)
28:      END;
29:      DO WHILE change >= 1;                   /*pennies*/
30:          CALL next$money(1);
31:      END;
32:      DO WHILE change >= 8;
33:          CALL next$money(0)                  /*zero out rest of coins*/
34:      END;
35: END make$change;
*
```

You can use the Print command to display the entire buffer or just pieces of the text. Type a Print command any time you wish to check alterations to the text. Experiment with different line numbers and the Print command (P).

Examples:

Type the following examples indicated by underlining.

```
*1,3P
```

EDIT displays lines 1 through 3.

```
1:  make$change:
2:  DO;
3:      DECLARE money(8) BYTE;                  /*this is the result*/
*
```

If you enter

```
*10,13P
```

EDIT displays lines 10 through 13.

```
10:             PROCEDURE(X);
11:                 I = I+1;
12:                 change = change - X;
13:             END
```

Deleting

Line 5 of the text is an unnecessary duplication of line 4. Delete the line as follows:

```
*5D
```

Delete has the same general form as Print, which allows you to delete multiple lines of the text as well as single lines. When you delete one or more lines, EDIT renumbers the remaining lines. Print lines 4, 5, and 6 so that you can verify that EDIT did renumber the lines.

```
*4,6P
4:      DECLARE change BYTE;                    /*number to be converted*/
5:
6:      next$money:                             /*this is a procedure*/
```

Inserting

Some text between lines 4 and 5 has been omitted. To insert the text type:

```
*5I
5:      DECLARE I BYTE;                         /*index to money array*/
6:  .
*
```

The EDIT program automatically displays and adjusts the line numbers for you. You may insert any amount of text before the line indicated in the Insert command (I). As with the Append command, you must type a period after you have inserted the desired text.

Moving

Print lines 8 through 10 so you will be able to examine any errors closely.

```
*8,10P
```

EDIT answers with

```
8:              DECLARE X BYTE;             /*X is specified */
9:              money(I) = X;
10:             PROCEDURE(X);
*
```

Lines 8 and 9 should come after line 10. Use the Move command (M) to rearrange the lines.

```
*8,9M10
```

The general form for this command is:

```
starting-line-number, ending-line-number M after-this-line-number
```

Joining

Lines 13 and 14 of the buffer should be one line. The Join command (J) allows you to accomplish this using the same general form as Print.

```
*13,14J
```

Print lines 1 through 13 to see if your alterations are correct. Your display should look like this.

```
1:  make$change:
2:    DO;
3:      DECLARE money(8) BYTE;                  /*this is the result*/
4:      DECLARE change BYTE;                    /*number to be converted*/
5:      DECLARE I BYTE;                         /*index to money array*/
6:
7:      next$money:                             /*this is a procedure*/
8:          PROCEDURE(X);
9:              DECLARE X BYTE;                 /*X is specified */
10:                 money(I) = X;
11:                 I = I+1;
12:                 change = change - X;
13:         ENDnext$money;
*
```

Notice that EDIT has resequenced the line numbers for you. The EDIT program will resequence the line numbers after each command in which resequencing is necessary. You should print frequently after your alterations so that you can keep track of any changing line numbers.

Using Line Numbers

Now that lines 1 through 13 look fairly error free, you can concentrate on the next group of lines. Display the buffer again if necessary. If for some reason you lose track of the current line, type a period after the prompt. This causes the current line to be displayed.

```
*.
```

In this case EDIT responds with line 13. This is the current line because the preceding command (Print) ended with this line.

```
13:         ENDnext$money;
```

The period may be used in place of line numbers in most commands. It specifies the current line in the buffer. Arithmetic offers another alternative to line numbers. You can use plus (+) or minus (-) signs in conjunction with numbers to take the place of line numbers. For example, +5 refers to the the line which is 5 lines beyond the current line. The number -8 refers to the line which is 8 lines before the current line.

Try using the period and some arithmetic to display the current line and the next 5 lines.

```
*.,+5P
```

EDIT displays the current line (13) plus the next 5 lines (14-18) and moves the marker to the last line printed.

```
13:         ENDnext$money;
14:
15:         change = 100 -                      /*wriye the cost here*/
16:         I = 0;                              /*initialize the index index*/
17:         IF change >= 50 THEN                /*half dollar*/
18:             CALL next$money(50);
*
```

If you execute the same command a second time, the results will be different.

```
*.,+5P

18:             CALL next$money(50);
19:         END;
20:         DO WHILE change >= 25;              /*quarters*/
21:             CALL next$money(25);
22:         DONE WHILE change >= 10;            /*dimes*/
23:             CALL next$money(10);
*
```

EDIT responds differently this time because the period, acting as a marker, was at line 18 after the first execution of the command. For this reason, EDIT prints the current line (18) and the next 5 lines (19-23) in response to the second execution of the command.

If you leave the comma out of a command that uses both arithmetic and a period you will get results very different to those you obtained in the ".,+5P" command. Type the following Print command.

```
*.+4P
```

The EDIT program displays the following line:

```
28:             END;
```

You can see that it makes a big difference if the comma is left out of this type of command. Instead of printing the current line and the next five lines, EDIT responded by moving the marker 4 lines ahead and printing the line.

You can also use arithmetic alone or with line numbers. If you had used the previous example without the period you would have gotten the same results. Try the following examples.

Enter

```
*-8P
```

EDIT answers with

```
20:             DO WHILE change >= 25;                  /*quarters*/
```

EDIT's period or marker was on line 28. The previous command directed the EDIT program to subtract 8 from the current line and to print that line.

Type

```
*12+3P
```

EDIT replies with line 15 because it is the line which equals 12 plus 3.

```
15:             change = 100 -     ;                    /*wriye the cost here*/
```

Substituting

The general form for the Substitute command (S) is as follows.

```
starting-line-number,ending-line-number S/wrong-word-or-phrase/replacement-word-or-phrase/
```

EDIT uses the first character after the "S" as a delimiter. EDIT reads this character as a delimiter and matches the word-or-phrase until it finds another of this same character. This allows you to replace the slashes with any other character. You might want to use another delimiter if the word-or-phrase contains a slash. However, to avoid confusion, this tutorial will use only the slash as a delimiter.

Print the buffer to line 23.

```
*1,23P

1:  make$change:
2:    DO;
3:      DECLARE money(8) BYTE;                  /*this is the result*/
4:      DECLARE change BYTE;                    /*number to be converted*/
5:      DECLARE I BYTE;                         /*index to money array*/
6:
```

```
7:      next$money:                                  /*this is a procedure*/
8:          PROCEDURE(X);
9:              DECLARE X BYTE;                      /*X is specified */
10:                 money(I) = X;
11:                 I = I+1;
12:                 change = change - X;
13:         ENDnext$money;
14:
15:         change = 100 -    ;                      /*wriye the cost here*/
16:         I = 0;                                   /*initialize the index index*/
17:         IF change >= 50 THEN                     /*half dollar*/
18:             CALL next$money(50);
19:         END;
20:         DO WHILE change >= 25;                   /*quarters*/
21:             CALL next$money(25);
22:         DONE WHILE change >= 10;/*dimes*/
23:             CALL next$money(10);
*
```

There is a typographical error on line 15. The word "write" in the comments is misspelled. Use the Substitute command (S) to deal with this error. You may wish to add the Print option to the end of the Substitute command so that you can check the change immediately.

```
*15S/wriye/write/P
```

EDIT performs the substitution and displays the corrected line.

```
15:         change = 100 -    ;                      /*write the cost here*/
```

If you examine the previously printed buffer, you will notice that line 13 has a spacing error. Invoke the Substitute command to correct the problem.

```
*13S/ENDnext/END next/P
```

EDIT displays the corrected version because the Print option is included.

```
13:         END next$money;
*
```

The Substitute command can also be used to delete a word or phrase within a single line or multiple lines. You can do this by substituting a "nothing" for the extra word or phrase. Line 16 has an unnecessary duplication of the word "index" in the comments. You can correct this mistake by using the Substitute command. Be sure to include the space before index so you do not end up with an extra space between "the" and "index."

```
*16S/ index//P
```

The double slashes cause EDIT to substitute "nothing" for the first occurrence of "(space)index". This, in effect, deletes "index" from the line. EDIT then displays:

```
16:         I = 0;                                   /*initialize the index*/
```

Changing

Line 17 should be a DO WHILE statement rather than an IF THEN statement. The Change command (C) can correct this more quickly than the Delete and Insert commands you used earlier. The Change command is actually a combination of the Delete and Insert commands.

```
*17C
17:         DO WHILE change >= 50                    /*half dollar*/;
18: .
```

You see that Change has the same general form as Print, Delete, and Insert. It also uses the period as do Append and Insert to inform EDIT that the operation is finished.

Text Copying

The next mistake is on line 22. The END statement for "quarters" is missing. You could correct this by inserting an END statement after line 21 or you could copy another END statement from elsewhere in the program. Copying is similar to moving except that the lines you indicate will not be moved, only duplicated. The general form for Text Copy (T) is the same as the general form for Move (M).

```
starting-line number, ending-line-number T after-this-line-number
```

Line 19 is an END statement. Copy line 19 to the line after 21.

```
*19T21
```

Now, Print the corrected part of the program to verify that the Text Copy command worked.

```
*1,22P

1:   make$change:
2:     DO;
3:        DECLARE money(8) BYTE;                    /*this is the result*/
4:        DECLARE change BYTE;                      /*number to be converted*/
5:        DECLARE I BYTE;                           /*index to money array*/
6:
7:        next$money:                               /*this is a procedure*/
8:             PROCEDURE(X);
9:                  DECLARE X BYTE;                /*X is specified */
10:                      money(I) = X;
11:                      I = I+1;
12:                      change = change - X;
13:            END next$money;
14:
15:       change = 100 -    ;                       /*write the cost here*/
16:       I = 0;                                    /*initialize the index*/
17:       DO WHILE change >= 50;                    /*half dollar*/
18:            CALL next$money(50);
19:       END;
20:       DO WHILE change >= 25;                    /*quarters*/
21:            CALL next$money(25);
22:       END;
*
```

Writing The Program Into A File

Suppose you wish to stop editing for a while but you also want to save the text contained in the EDIT buffer. In order to save the text you must write it into a file on secondary storage. The general form for writing something into a file is:

```
W file-name
```

This manual will use an iRMX 86 file name for the sake of simplicity. Your file name may be different. Refer to the *iRMX 86 HUMAN INTERFACE REFERENCE MANUAL* for a description of iRMX 86 file names.

You may call the file anything you want but this manual will refer to it as "program/money.plm". You can use the Write command (W) to place the contents of the buffer on secondary storage.

```
*W program/money.plm
```

EDIT responds with:

```
program/money.plm: 35 lines, 929 bytes.
*
```

Quitting EDIT

The Quit command (Q) permits you to exit the EDIT program. If you quit EDIT before you have written the text into a file, you will delete all the text and the corrections to the text you have added during the editing session. The Quit command is always written as a single letter "Q".

```
*Q
```

The EDIT program will no longer be in effect and your system will respond with its normal prompt.

Re-Entering EDIT

You could re-enter EDIT just as you originally invoked the program.

```
*ED
```

This way of entering EDIT doesn't automatically access the file "program/money.plm". To begin editing "program/money.plm" again you must use the Edit command (E). The general form for this command is the same as the general form for Write.

```
E file-name
```

If you were to enter the previous commands with "program/money.plm" as the file, EDIT would answer as follows:

```
program/money.plm: 35 lines, 929 bytes.
*
```

You could then begin editing again. However, a simpler way to re-enter or to enter EDIT when you know the name of the file you wish to alter is to type:

```
*ED file-name
```

Invoke EDIT and specify the file "program/money.plm" at the same time.

```
*ED program/money.plm
```

EDIT gives the same response it would have given had you invoked EDIT and entered the file the long way.

```
program/money.plm:  35 lines, 929 bytes.
*
```

Print the buffer.

```
*1,$P
```

EDIT displays the contents of the buffer.

```
make$change:
    DO;
        DECLARE money(8) BYTE;              /*this is the result*/
        DECLARE change BYTE;                /*number to be converted*/
        DECLARE I BYTE;                     /*index to money array*/

        next$money:                         /*this is a procedure*/
            PROCEDURE(X);
                DECLARE X BYTE;             /*X is specified */
                money(I) = X;
                I = I+1;
                change = change - X;
        END next$money;
        change = 100 -    ;                 /*write the cost here*/
        I = 0;                              /*initialize the index*/
        DO WHILE change >= 50;              /*half dollar*/
            CALL next$money(50);
        END;
```

```
        DO WHILE change >= 25;              /*quarters*/
            CALL next$money(25);
        END;
        DONE WHILE change >= 10;            /*dimes*/
            CALL next$money(10);
        END;
        DO WHILE change >= 15;              /*nickels*/
            CALL next$money(5)
        END;
        DO WHILE change >= 1;               /*pennies*/
            CALL next$money(1);
        END;
        DO WHILE change >= 8;
            CALL next$money(0)              /*zero out rest of coins*/
        END;
    END make$change;
*
```

Notice that the lines are no longer numbered. The Numbering command (N) is exclusively an EDIT feature. They are not part of the file and when you wrote the text into the file the numbers were not transferred. Of course, you could renumber the lines with the Number command (N), but it is not necessary for this editing session.

Searching

A Search command automatically looks through the text to locate the first instance of the a word or phrase which you indicate. The direction and starting line of the Search depend on the specific Search command. If you should search past the beginning or end of the buffer, EDIT "wraps around" and continues to search for the word-or-phrase in the direction you specify.

Forward Searching

The EDIT Forward Search looks through the text for the word or characters which you indicate. It begins at the line after the current line and searches toward the end of the file. This command consists of words or phrases enclosed in slashes (/). You *must* use slashes around the word-or-phrase in the Forward Search command. For example if you want to locate the first line which contains a DECLARE statement you would type:

```
*/DECLARE/
```

In this case EDIT answers with:

```
        DECLARE money(8) BYTE;              /*this is the result*/
*
```

If you want to repeatedly search for a particular word, you would type double slashes after each instance of the word that search finds. Let's say you want to search for the word "change" more than one time. Type:

```
*/change/
```

EDIT replies with the next line that contains "change."

```
        DECLARE change BYTE;                /*number to be converted*/
*
```

If you want to continue to search for the next occurrence of "change" you should type a double slash (//).

```
*//
```

Since EDIT remembers the last word or phrase, it responds with the next line that contains "change."

```
                change = change - X;
*
```

Forward Searching With Commands

These enclosed words or phrases can be used in place of line numbers in all commands that use line numbers. If you examine the previously displayed buffer you will see that the line identified as "dimes" is in error. The word DONE should be the word DO. Correct this with a Search and the Substitute command (S).

```
*/dimes/S/DONE/DO/
*
```

EDIT searched for a line containing dimes and it then changed DONE to DO in that line.

Display the corrected version of the line.

```
*P
```

EDIT replies:

```
        DO WHILE change >= 10;          /*dimes*/
```

Reverse Searching

The EDIT Reverse Search works very much like the Forward Search. It looks through the text for the word which you indicate. However, it begins at the line before current line and searches toward the beginning of the file. This command consists of words or phrases enclosed in question marks (?). Try the same examples for Reverse Search that you used for Forward Search and compare the results.

Search, in reverse order, for the first occurrence of the word DECLARE

```
*?DECLARE?
```

EDIT responds with

```
        DECLARE X BYTE;          /*X is specified */
```

This is the first line containing DECLARE that EDIT finds when looking in a reverse direction from the current line.

If you want to continue the Reverse Search you would type double question marks (??) just as you typed the double slash marks for a continuing Forward Search.

```
*??
```

EDIT answers with the next line backward which contains the word DECLARE.

```
        DECLARE I BYTE;          /*index to money array*/
```

Global Editing

A Global Command searches out *each* instance of a word or phrase you specify. You can use the Global command (G) in combination with one or more EDIT commands. The EDIT commands perform their normal functions on a global or entire-file basis.

The general form for the Global command (G) is a bit more complex than the previous general forms. A Global command can precede any other EDIT command except another Global command.

```
starting-line-number, ending-line-number G/word-or-phrase/command
```

Suppose you want to display all the lines between the marker and the end of the file that contain the word BYTE. Normally you would enter many commands to do this but with global editing you need only one command.

```
*G/BYTE/P       (G search-command Print-command))
```

EDIT responds with all lines that contain the word BYTE.

```
DECLARE money(8) BYTE;              /*this is the result*/
DECLARE change BYTE;                /*number to be converted*/
DECLARE I BYTE;                     /*index to money array*/
        DECLARE X BYTE;             /*X is specified */
```

The PL/M program you are altering deals with money, but "coins" is a more descriptive term. Change every occurrence of "money" to "coin" with a combination of the Substitute and Global commands.

```
*G/money/S//coin/GP
```

EDIT finds every line that contains, "money", substitutes the word "coin" for "money", and prints the corrected lines.

```
DECLARE coin(8) BYTE;               /*this is the result*/
DECLARE I BYTE;                     /*index to coin array*/
next$coin:                          /*this is a procedure*/
        coin(I) = X;
END next$coin;
    CALL next$coin(50);
    CALL next$coin(25);
    CALL next$coin(10);
    CALL next$coin(5)
    CALL next$coin(1);
    CALL next$coin(0)               /*zero out rest of coins*/
*
```

Print the buffer to remind yourself of the corrections you have already made and to spot any additional problems.

```
make$change:
    DO;
        DECLARE coin(8) BYTE;       /*this is the result*/
        DECLARE change BYTE;        /*number to be converted*/
        DECLARE I BYTE;             /*index to coin array*/

        next$coin:                  /*this is a procedure*/
            PROCEDURE(X);
                DECLARE X BYTE;     /*X is specified */
                coin(I) = X;
                I = I+1;
                change = change - X;
        END next$coin;

        change = 100 -    ;         /*write the cost here*/
        I = 0;                      /*initialize the index*/
        DO WHILE change >= 50;      /*half dollar*/
            CALL next$coin(50);
        END;
        DO WHILE change >= 25;      /*quarters*/
            CALL next$coin(25);
        END;
        DO WHILE change >= 10;      /*dimes*/
            CALL next$coin(10);
        END;
        DO WHILE change >= 15;      /*nickels*/
            CALL next$coin(5)
        END;
```

```
        DO WHILE change >= 1;                    /*pennies*/
            CALL next$coin(1);
        END;
        DO WHILE change >= 8;
            CALL next$coin(0)                    /*zero out rest of coins*/
        END;
    END make$change;
*
```

Changing The File Name

Now that the text has been corrected, the file "program/money.plm" does not seem appropriately named. You can use the File Name command (F) to either display the current file name or to change it. If you type

```
*F
```

EDIT will display the file name it remembers.

```
program/money.plm
```

But, if you type

```
*F program/coin.plm
```

EDIT changes the remembered file name and displays the new file name.

```
program/coin.plm
```

If you have written the file into memory using the old name, you will still have a file on secondary storage with the old name.

Exiting EDIT

The Exit command (X) is a combination of the Write command (W) and the Quit command (Q). You must not use this command unless EDIT remembers a file name or you explicitly assign a file name to your text with the Exit command. This command writes the contents of the buffer into the last file EDIT remembers or the name you picked for the file. It then quits the EDIT program.

Exit EDIT.

```
*X
```

Your system responds with the last file name it remembers.

```
program/coin.plm:  35 lines, 929 bytes.
```

The EDIT program will no longer be in effect and your system will respond with its normal prompt.

CHAPTER 3
INVOKING EDIT

You know that you can invoke the EDIT program by typing

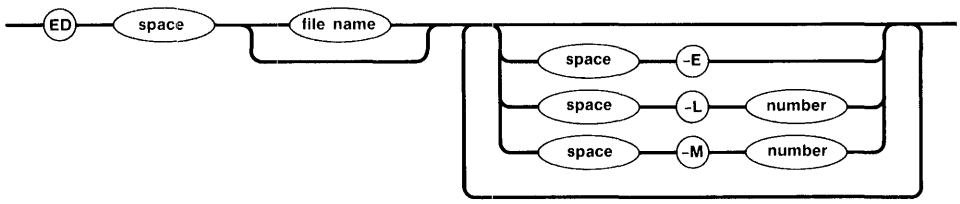
```
*ED
```

You also know that you can specify a file name when you invoke the EDIT program if you type:

```
*ED file-name
```

There are additional options, called controls, available to you when you invoke EDIT. This chapter provides a pictorial syntax of all possible invocations along with a description of each control.

Controls allow you to change the defaults programmed into EDIT. EDIT has 4 controls which change such things as the number of lines permitted in the buffer (-L), the number of bytes to hold macro definitions (-M), the echoing of command file input (-E), and whether there is a file name. If you want to change the defaults, you can type one of the previously mentioned controls when you invoke EDIT. You must specify a decimal number with two of the controls.



File Name

The File Name control allows you to specify a file name when you invoke the EDIT program. If the file already exist on secondary storage, EDIT reads the contents of the file into the buffer. If the file does not exist, EDIT uses the file name to label whatever you enter in the buffer. EDIT remembers this name so you do not have to enter it when you want to write the file onto secondary storage.

Echo Control

You can cause the EDIT program to echo any command file input when you use the -E control. Command file input is any command which you do not type directly on the terminal. The -E control causes EDIT to show you (echo) the automatic input from these files on your terminal. When you use the -E control, EDIT will also echo all commands performed by a special file called ED.MAC. Refer to Chapter 6 for a complete description of command files and ED.MAC.

Example:

Suppose you are ready to invoke the EDIT program and create a new file called "test.plm" on a device named ":f1:". You know that you will be using macros and command files extensively, and you want the automatic input displayed. Invoke EDIT, specifying the file name and -E controls.

```
ED :f1:test.plm -E
```

EDIT responds by displaying all the contents of ED.MAC and

```
:f1:test.plm: new file
```

Line Control

The -L control allows you to specify the size (in lines) of the buffer in which EDIT stores text. The EDIT program has a default buffer of 3000 lines. You can increase or decrease this limit with the -L control.

Example:

Suppose you have a file called "program/money.plm" that is going to go on device ":f1:". If you know that this file is going to take at least 3050 lines you can use the -L control to expand EDIT's buffer capacity.

```
ED :f1:program/money.plm -L4000
```

EDIT will respond with a sign-on (not shown), the new file name, and a prompt.

```
:f1:program/money,plm: new file.
*
```

Although EDIT does not respond visibly to the -L control, it does allow for the extra lines.

NOTE

Even if you allow extra lines, you can overflow the buffer. Every time you alter lines of text, you consume EDIT buffer space because EDIT does not reclaim this space. If you use more buffer space than EDIT allows, EDIT creates temporary files on secondary storage to hold the text. EDIT will become slow in responding to some commands when this happens. To remedy the situation, write the program into secondary storage and use the edit command (E) to re-enter the file.

Macro-Space Control

The macro-space control (-M) allocates bytes to hold macro definitions. The EDIT program has a default of 1024 bytes specifically to hold macro definitions. You can increase or decrease this number with the -M control.

Example:

You can use the -M control to give yourself more "space" for macro definitions. The following line tells EDIT to allow 2000 bytes of space for macro definitions.

```
ED -m2000
```

As with the -L control, EDIT responds only with a sign-on (not shown), and a prompt.

CHAPTER 4
SPECIAL CHARACTERS AND COMMANDS

This chapter describes the special characters and commands that EDIT recognizes. Special characters are those which are not in the alphabet and have a special meaning within the EDIT program. Special commands are commands which have a special meaning when you use them with a special character. They include:

- period (.)
- dollar sign ($)
- carriage return (c/r)
- forward search (/word-or-phrase/)
- reverse search (?word-or-phrase?)
- up arrow or circumflex (↑)
- square bracket ([])
- asterisk (*)
- ampersand (&)
- backslash (\ )

These characters can have different meanings to EDIT, depending upon the way in which you use them. This chapter describes each character in detail and gives examples of their functions in varied situations.

Period

The period (.) represents the current line of the text. When you use the period by itself, EDIT responds by displaying the current line of the file. In general, you can substitute a period in place of line numbers in EDIT commands.

Examples:

Suppose your file contains the following lines of text. The second line happens to be the current line.

```
END NEXT$coin;
    CALL next$coin (50);
    CALL next$coin (25);
    CALL next$coin (10);
```

If you type

```
*  .
```

EDIT answers with the current line, which in this case is

```
    CALL next$coin (50);
```

You can use the period to display the current line and the next 2 lines by typing

```
*.,+2P
```

EDIT replies with

```
    CALL next$coin (50);
    CALL next$coin (25);
    CALL next$coin (10);
```

Matching Using The Period

You have already used the period (.) as a marker for the current line. This character does have another application. You can use the period to represent any character to be matched in a substitute, global, or search command. Suppose your file contains the following text.

```
        PROCEDURE(X);
          DECLARE X BYTE;                    /*X is specified when
          coin(I) = X;                       procedure is called*/
          I = I+1;
          change = change - X;
END next$coin;
```

Type the following line:

```
*G/I.1/P
```

The period between the "I" and the "1" means any character. The previous example searches for all three-character strings that have a first character "I" and a last character"1".

However, only one statement matches in your file.

```
I = I+1;
```

If you had wanted to look for a longer string, with "I" as the first character, and "1" as the last character, you could have used additional periods for the middle characters.

Dollar Sign

The dollar sign ($) represents the last line of the text. When you use the dollar sign by itself, EDIT responds by displaying the last line of the buffer. You can also use the dollar sign in place of a line number in most EDIT commands.

Examples:

Suppose your file consists of the following lines.

```
DO WHILE I < 8;
     CALL next$coin (5);
END;
```

If you type

```
*$
```

EDIT will display the last line of the buffer.

```
END;
```

Use the dollar sign ($) and the print command (P) to display the entire buffer.

```
*1,$P
```

EDIT answers with

```
DO WHILE I < 8;
     CALL next$coin (5);
END;
```

Dollar Sign — End of Line

You already know that the dollar sign can represent the end of the file or buffer. The dollar sign ($) also signifies the end of a line. You can use this character with a

forward or reverse search, a substitute command, global command, or a "V" command (refer to Chapter 5 for a description of the "V" command).

When you use a forward or reverse search with the dollar sign, EDIT finds the line which ends with the word or phrase you specified.

Example:

Suppose your file contains the following lines.

```
next$coin:
PROCEDURE (X);
DECLARE X BYTE;
coin (I) = X + coin
I = I+1;
```

Substitute the second occurrence of "coin" in line four with the word "change" and a semicolon. Print the altered line.

```
*4S/coin$/change;/P
```

EDIT answers with

```
coin (I) = X + change;
```

Carriage Return

You can use the carriage return as a special character. When you use it by itself, the carriage return allows you to step through a file one line at a time. This character causes EDIT to display the next line of the file each time you type it.

Suppose your file consists of the following lines and your marker is at line one.

```
DO WHILE I < 8
    CALL next$coin (5);
END
```

If you type

```
*(c/r)
```

EDIT answers

```
        CALL next$coin (5);
```

If you enter another carriage return, EDIT responds with the next line of the file.

```
END;
```

Forward Search

The forward search (/word-or-phrase/) is a special command. It begins at the line after the current line and looks through the text in a forward direction for the line which contains the word-or-phrase you specify. If you search past the end of the text, this command wraps around to the beginning of the file. If the word-or-phrase is not in the text, EDIT responds with an error message (?).

You can use the forward search by itself to find the first occurrence of the line which contains the word-or-phrase you typed in the slashes. If you want to find the next line which contains the word-or-phrase, you can type double slashes (//) after the EDIT prompt.

In general, you can use a forward search in place of a line number in EDIT commands.

Examples:

Let's say your file consists of the following lines of text.

```
make$change:
    DECLARE money(8) BYTE;
    DECLARE change BYTE;
```

Do a forward search for the word "change".

```
*/change/
```

EDIT responds with

```
make$change:
```

Use the double slashes to search for the next occurrence of "change".

```
*//
```

EDIT answers with the next line which contains the word "change".

```
    DECLARE change BYTE;
```

Reverse Search

The reverse search (?word-or-phrase?), like the forward search, is a special command. It begins at the line before the current line and looks through the text in a reverse direction for the line which contains the word-or-phrase you specify. If you search past the beginning of the text, this command wraps around to the end of the file. If the word-or-phrase is not in the text, EDIT responds with an error message (?).

When you use the reverse search by itself, EDIT searches backward and finds the first occurrence of the word or phrase you typed in the question marks. If you want to find the next line which contains the word or phrase, you can type double question marks (??) after the EDIT prompt.

In general, you can use a reverse search in place of a line number in EDIT commands.

Examples:

Suppose your file is made up of the following text and your marker is at line one.

```
make$change;
    DECLARE money(8) BYTE;
    DECLARE change BYTE;
```

Do a reverse search for the word "change".

```
*?change?
```

EDIT responds with the next previous line which contains the word.

```
    DECLARE change BYTE;
```

Use the double question marks (??) to search for the next occurrence of "change".

```
*??
```

EDIT answers with the following line.

```
make$change;
```

Up Arrow

The up arrow (↑), which on some terminal is shown as a circumflex (^), signifies the beginning of a line. You can use this character as an address or with a Forward or Reverse Search, or a Substitute, Global, Move, Text Copy, and "V" command (refer to Chapter 5 for a description of the "V" command).

When you use a forward or reverse search with the up arrow, EDIT finds the line which begins with the word or phrase you specify. An up arrow is equivalent to a minus sign (-) when you use it as an address.

Example:

Suppose your file contains the following lines.

```
next$coin:
PROCEDURE (X);
DECLARE X BYTE;
coin (I) = X;
I = I+1;
```

Print the line which contains "coin" as its first word.

```
*/↑coin/ P
```

EDIT answers with the following line.

```
coin (I) = X;
```

When you use the up arrow (↑) with a substitute or global command, EDIT finds the word-or-phrase only if it is at the beginning of the line.

Example:

Suppose your file contains the following lines.

```
next$coin:
PROCEDURE (X);
DECLARE X BYTE;
coin (I) = coin + X;
I = I+1;
```

Substitute the first occurrence of "coin" in line four with the word "change" and print the altered line.

```
*4S/↑coin/money/P
```

EDIT answers with

```
money (I) = coin + X;
```

Square Brackets

You can use square brackets to denote a set of characters. If any character in the brackets is the same as any character in the file, EDIT displays that line. You can use these characters with a forward or reverse search, a substitute command, a global command, or a "V" command (refer to Chapter 5 for a description of the "V" command).

When you use a forward or reverse search with the square bracket, EDIT finds the line which contains the any character you typed inside the brackets.

Example:

Suppose your file contains the following lines of text.

```
END next$coin;
     CALL next$coin (50);
     CALL next$coin (25);
     CALL next$coin (10);
```

If you type

```
*/[12]/
```

EDIT will find the "2" and display the following line.

```
     CALL next$coin (25);
```

If you repeat the forward search request

```
*//
```

EDIT will display

```
     CALL next$coin (10);
```

because it found the "1".

When you use the square brackets with a substitute or global command, EDIT finds the line or lines which contain any of the characters inside the brackets.

Example:

Suppose your file contains the following lines of text.

```
END next$coin;
     CALL next$coin (50);
     CALL next$coin (25);
     CALL next$coin (10);
```

Substitute every occurrence of the numbers "0" and "5" with the number "1" using the square brackets with the substitute and global commands.

```
*G/[05]/S/[05]/1/G
```

Print the buffer to check your corrections.

```
*1,$P
END next$coin;
     CALL next$coin (11);
     CALL next$coin (21);
     CALL next$coin (11);
```

Now, suppose your file contains the following lines of text.

```
CALL next coin (50)
CALL nexttcoin (60)
CALL next coin (70)
```

You can use the square brackets as part of a pattern to change "next coin" and "nexttcoin" to "next$coin". Type

```
*G/xt[ t]co/S/xt[ t]co/xt$co/GP
```

The previous command tells EDIT to find all lines containing either "xt co" or "xttco" and change both cases to "xt$co". EDIT answers with the corrected lines in response to the Global and Print options.

```
CALL next$coin (50)
CALL next$coin (60)
CALL next$coin (70)
```

Suppose your file contains the following lines.

```
make$change:
    DO;
            DECLARE money(8) BYTE;          /*this is the result*/
            DECLARE change BYTE;            /*number to be converted*/
            DECLARE I BYTE;                 /*index to money array*/
```

You can use the square brackets and the Up Arrow (↑) to represent "anything but the contents of these brackets." Suppose you want to print all statements that do *not* contain the word DECLARE. Type

```
*G/[↑DECLARE]/GP
```

EDIT answers with

```
make$change:
    DO;
```

Asterisk

The asterisk (*) means "any number of." For example "X*" means any number of X's. This character is useful for deleting parts of a line when you use it in combination with the substitute command.

Examples:

Suppose your file consists of the following lines.

```
DO WHILE change = 10;               /*dimes*/
CALL next$coin(10);
END;
```

Suppose you want to delete line 1 up to the word "change." You can do this by using the period (.), the asterisk (*), and the substitute command. You already know that the period matches anything and the asterisk means "any number of" so the following command means, "in line 1, delete any number of characters up to and including WHILE."

```
*1S/.*WHILE//
```

If you print the file you can see exactly what this command did.

```
change >= 10;                       /*dimes*/
CALL next$coin(10);
END;
```

It deleted everything up to the space before the word "change".

Ampersand

You can use the ampersand (&) in two ways. Either way saves typing time. An ampersand can represent the word or phrase you chose to change in a substitute command.

Suppose your file contains the following lines.

```
1:  make$change:
2:      DO;
3:          DECLARE coin(8) BYTE;       /*this is the result*/
4:          DECLARE change BYTE;        /*number to be converted*/
5:          DECLARE I BYTE;             /*index to coin array*/
```

Suppose you want to put parentheses around the "make$change" statement. You could retype the statement or you could use the ampersand along with the period and the asterisk.

```
*1s/.*:/(&)/P
```

The previous line tells EDIT to find everything in line 1 up to and including the colon, put parentheses around it, and print it. The ampersand represents the word or phrase you specified in the substitute command. So, EDIT responds with the altered line.

```
1:    (make$change:)
```

The ampersand (&) is also a short-hand way of saying, "do this operation on the lines which were previously addressed."

For example, suppose your file consists of the following lines.

```
1:    make$change:
2:        DO;
3:            DECLARE money(8) BYTE;          /*this is the result*/
4:            DECLARE change BYTE;            /*number to beconverted*/
5:            DECLARE I BYTE;                 /*index to money array*/
6:
7:            next$money:                     /*this is a procedure*/
8:              PROCEDURE(X);
9:                DECLARE X BYTE;             /*X is specified */
```

Print lines 2 and 3.

```
*2,3P
```

EDIT answers with

```
2:    DO;
3:            DECLARE money(8) BYTE;          /*this is the result*/
```

Now suppose you want to delete the previously printed lines. All you have to do is to use the ampersand (&) and the delete command (D) because EDIT remembers the addresses of the last command.

```
*&D
```

Print the buffer to verify that the ampersand did actually cause EDIT to execute the delete command on the previously addressed lines.

```
*1,$P
```

EDIT replies with

```
1:    make$change:
2:            DECLARE change BYTE;            /*number to beconverted*/
3:            DECLARE I BYTE;                 /*index to money array*/
4:
5:            next$money:                     /*this is a procedure*/
6:              PROCEDURE(X);
7:                DECLARE X BYTE;             /*X is specified */
```

Backslash

If you try to substitute for special characters, EDIT will respond with an error message (?). Since there are times when you may need to alter these characters, the EDIT program includes a character which takes away all special meaning from the character. This character is the backslash(\). If you type a backslash before a special character you wish to change, you can alter it as you would any other character.

Examples:

Suppose your file contains the following lines.

```
DO WHILE change >= 1;                    /*pennies*/
   CALL next&coin(1);
END;
```

The word "next&coin" should read: next$coin. Since the ampersand has a special meaning in the EDIT program, you must use the backslash (\) in order to correct this line. Use the substitute command and the backslash character (\) to fix this error.

```
*S/next\&coin/next$coin/P
```

EDIT answers with the corrected line.

```
      CALL next$coin(1);
```

Now, suppose your file contains the following lines.

```
9:          DECLARE X BYTE;              /*X is specified */
10:            money(I) = X;
11:            I = I+1;
12:            change = change - X;
13:      ENDnext$money;
```

If you want to split line 13 between END and "next" into two separate lines. You can do this by using the backslash to take away the special meaning of a carriage return. Normally after you type a carriage return, EDIT tries to execute a command. The following example has the carriage return displayed for illustration only.

```
*13s/END next/END\ (c/r)
next/
```

Print lines 13 and 14 to see that the command was executed correctly.

```
*13,14P
```

EDIT answers

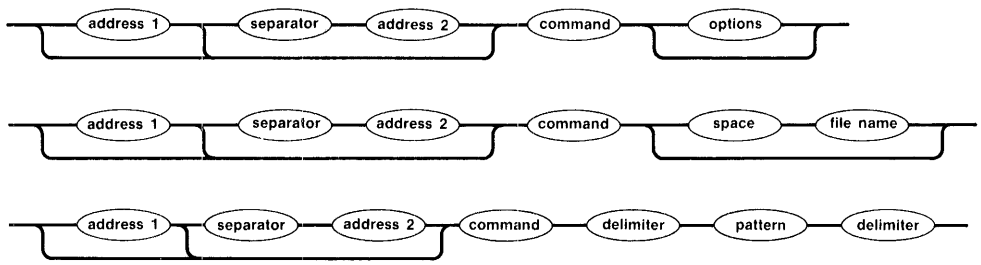
```
13:          END
14:  next$money;
```

CHAPTER 5
COMMANDS

This chapter consists of three parts. It begins with some introductory material followed by a command dictionary. The balance of the chapter contains detailed command descriptions. The introductory material will help you to understand the detailed descriptions and pictorial syntax of the commands in the tabbed section. The commands are arranged in alphabetical order with one command on each tabbed page so you can refer to them easily.

Common Command Syntax

Here are three examples of typical command syntax. The following sections explain each of the syntax elements.



Addresses

Addresses are a way of specifying one or more lines of text. EDIT uses many types of addresses with the line number being the most common.

NOTE

If you make an explicit reference to line number zero (0) EDIT will usually respond with an error message. However, zero is an acceptable address for the Append command and the Read command.

EDIT finds everything between the first address and the second address and performs the command. You can use any of these wherever a command syntax calls for an address.

no address	If you type a command with no address, EDIT will execute the command on the current linc or set of lines.
line number	EDIT finds the line number you type.
period (.)	The period directs EDIT to find the current line.
dollar sign ($)	The dollar sign ($) tells EDIT to find the last line of the buffer.

plus sign (+) options

When you address a line with a plus sign (+), EDIT begins at the current line and adds the number of plus signs to find the address.

When you address a line with a plus (+) sign and an integer, EDIT begins at the current line and adds the integer to find the address.

You can use any "plus sign option" with any address. EDIT begins at the address and adds the number of lines.

minus sign (-) options

When you address a line with a minus sign (-), EDIT begins at the current line and subtracts the number of minus signs to find the address.

When you address a line with a minus (-) sign and an integer, EDIT begins at the current line and subtracts the integer to find the address.

You can use any "minus sign option" with any address. EDIT begins at the address and subtracts the number of lines.

'letter

When you address a line with a quote sign and a letter, EDIT looks for a line you marked with the "K" command and the letter (see the "K" command).

pattern

A pattern is a sequence of characters, word, or phrases that you choose for EDIT to match. This manual calls a pattern a "word-or-phrase" in previous chapters to avoid confusion. When you use a pattern as an address, you must enclose it in either slashes (/) for Forward Searches, or question marks (?) for Reverse Searches. These enclosing characters are called "delimiters."

/pattern/

EDIT does a Forward Search for the line which contains the pattern you type within the slashes (//). Remember, the forward search begins after the current line.

?pattern?

EDIT does a Reverse Search for the line which contains the pattern you type within the question marks (??). Remember, the Reverse Search begins before the current line.

period (.)

When you use a period as a part of a pattern, it represents any character.

dollar sign($)

If you use a dollar sign ($) with a pattern, EDIT finds the line containing the pattern only if it is the last thing on the line. You must type the dollar sign immediately preceding the *second* delimiter, or EDIT will treat it as a normal character.

up arrow (↑)

If you use an up arrow (↑) with a pattern, EDIT finds the line containing the pattern only if it is the first thing in a line. You must type the up arrow immediately *after* the *first* delimiter, or EDIT will treat it as a normal character.

asterisk (*)	You can use the asterisk (*) to represent "any number of " the previous character in a pattern.
[set-of-characters]	If you use a set of characters enclosed in square brackets as part of a pattern, EDIT finds the line containing *one* of the characters enclosed in the brackets.

Separators

The separator serves to divide the addresses. It can be either a comma or a semicolon. However, it is important to note that EDIT responds differently when you use a semicolon as a separator.

If you use a comma as a separator between addresses, EDIT looks for both lines beginning at the line after the current line. In contrast, when you use a semicolon as a separator, EDIT moves the marker to the first addressed line. EDIT then looks for the second addressed line from this line.

For example, suppose your file consists of the following lines.

```
make$change:
    DO;
        DECLARE coin(8) BYTE;          /*this is the result*/
        DECLARE change BYTE;           /*number to be converted*/
        DECLARE I BYTE;                /*index to coin array*/
```

Suppose your marker is at the first line of this text and you want to delete the first and second DECLARE lines. Try using the Delete command with a comma as a separator.

```
*/DECLARE/,/DECLARE/D
```

Print the buffer to verify your changes.

```
*1,$P
```

EDIT answers with

```
make$change:
    DO;
        DECLARE change BYTE;           /*number to be converted*/
        DECLARE I BYTE;                /*index to coin array*/
```

You can see that EDIT deleted only the first DECLARE statement. This is because the EDIT program found the first DECLARE statement for both addresses. Use the same text as in the first example (displayed below) and perform the same command using a semicolon rather than a comma as a separator.

```
make$change:
    DO;
        DECLARE coin(8) BYTE;          /*this is the result*/
        DECLARE change BYTE;           /*number to be converted*/
        DECLARE I BYTE;                /*index to coin array*/
```

```
*/DECLARE/;/DECLARE/D
```

Print the buffer to verify your results.

```
*1,$P
```

EDIT answers with the following lines.

```
make$change:
    DO;
        DECLARE I BYTE;                /*index to coin array*/
```

NOTE

This manual uses a comma for the separator in all cases. Keep in mind that you can use a semicolon wherever there is a comma.

Commands

Commands are the one-letter or one-character instructions to which EDIT is programmed to respond.

Options

Options are commands which can be added to other commands. These are specific to individual commands and they are explained in the detailed descriptions of each command.

File Name

This manual will use iRMX 86 file names for the sake of simplicity. These file names consist of a logical name for the device, and an iRMX path. Your file name may be different. Refer to the *iRMX 86 HUMAN INTERFACE REFERENCE MANUAL* for a description of paths used in naming iRMX 86 files.

Pattern

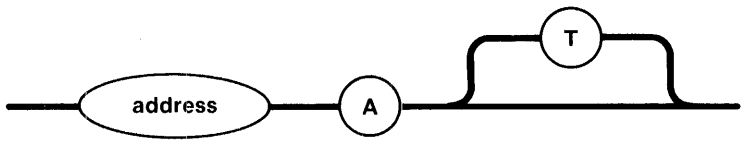
You can also use patterns on the right side of some commands. This is especially useful when you want EDIT to execute a command on a particular word, phrase, character, or string of characters rather than on an entire line. Patterns on the right side of a command behave exactly the same as the patterns described in the "Addresses" section of this chapter with one exception: EDIT reads the first character after the command as the delimiter so the slash (/) and the question mark (?) do not represent a Forward or Reverse Search.

For example, EDIT reads the pattern "1test1" as "delimiter pattern delimiter". It then tries to find an occurrence of "test".

Command Dictionary

A — Append Or Add To

The Append command (A) allows you to add text *after* a line. You can also create text to be written into a new file when you use the Apppend command with no address.



MARKER POSITION

After executing this command, EDIT moves the marker to the last line you added.

PARAMETERS

no address	EDIT finds the current line and waits for you to add text.
address	EDIT finds the address you choose and waits for you to add text. Remember, you **can** use zero as an address with the Append command.
T (tab)	When you use a Tab option (T) with the Append command, EDIT automatically begins your next line at the same tab setting you chose for the previous line. If you want to add another tab, simply press the TAB key. If you want to begin your next line at a lesser tab setting, type two periods (..) followed by a carriage return on an otherwise blank line. This will set the tab back one from the previous line. EDIT will not include the periods in the file whereas the tabs become part of the file.

You choose the location of the new text by means of the previously described parameters. After you have added all the text you wish, type a period (.) and a carriage return on a blank line to stop inserting.

Examples:

Assume that you have no text in the buffer. Append the following text to create a new file.

```
*A
DO WHILE i<8;
    CALL next$coin (5);
END;
.
*
```

EDIT adds this text to the buffer. Suppose you want to add another CALL statement after the existing CALL statement in the text you just created. Your marker is at the last line of the buffer, so use a reverse search as your address and append the following statement.

```
*?CALL?A
    CALL next$coin (0);
.
*
```

A — Append Or Add To (Continued)

Suppose you have a file that consists of the following lines.

```
1:    make$change:
2:        DO;
3:            DECLARE money(8) BYTE;          /*this is the result*/
4:            DECLARE change BYTE;            /*number to beconverted*/
5:            DECLARE I BYTE;                 /*index to money array*/
```

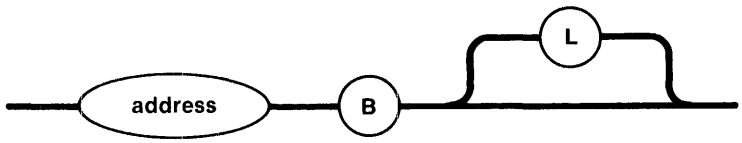
If you use the Append command with the Tab option, you can keep track of your spacing more easily. Add the following text using Append and the Tab option as shown. Tabs are shown by (T).

```
*5AT
6:            next$money:                     /*this is a procedure*/
7:              (T) PROCEDURE(X);
8:                (T) DECLARE X BYTE;         /*X is specified*/
9:                      money(I) = X;
10:               I = I=1;
11:               change = change - X;
12:               ..
12:             ..
12:           END next$money;
```

EDIT remembered the tab setting from line 5 and set line 6 to that setting.

B — Back One Screen

The "B" command (B) displays the current line of the file plus up to 22 previous lines. This command is equivalent to the Print command "-22,.P"; therefore it saves you keystrokes. As with the Print command, the lines are displayed in their original order.



MARKER POSITION

This command moves the marker to the addressed line.

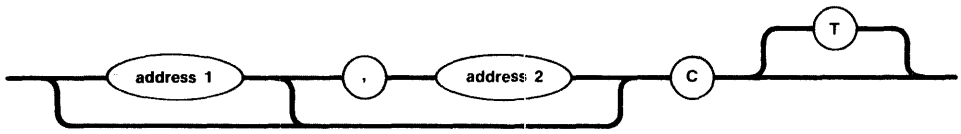
PARAMETERS

no address	EDIT finds the current line and displays it plus up to 22 previous lines.
address	EDIT finds the address you choose and displays that line plus up to 22 previous lines.
L (list)	When you use a List option (L) with the "B" command , EDIT displays *all* characters, including nonprintable ASCII characters, in the line you addressed plus up to 22 previous lines.

This command is more useful on a screen-type terminal than on a hard-copy or single-line terminal, since it displays the screen full of text preceding the line that you specify.

C — Change

This command permits you to change a line or several lines of text. The Change command (C) is equivalent to a combination of the Delete (D) and Insert (I) commands.



MARKER POSITION

After executing this command, EDIT moves the marker to the last line you entered.

PARAMETERS

no address	EDIT finds the current line and replaces it with the text that you type.
address 1	EDIT finds the first address you specify. If this is the only address you specify, EDIT replaces this line with the text that you type.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT deletes all text between the first address and the second address, inclusive. It then inserts the text that you type.
T (tab)	When you use a Tab option (T) with the Change command, EDIT automatically begins your next line at the same tab setting you chose for the previous line. If you want to add another tab, simply press the TAB key. If you want to begin your next line at a lesser tab setting, type two periods (..) followed by a carriage return on an otherwise blank line. This will set the tab back one from the previous line. EDIT will not include the periods in the file whereas the tabs become part of the file.

You choose the location of the new text by means of the previously described parameters. After you have replaced all the text you wish, type a period (.) and a carriage return on a blank line to stop changing.

Examples:

Suppose that part of your file looks like this.

```
20:   DO WHILE change >= 25;
21:     CALL next$money (25);
```

Alter it with the Change command as follows.

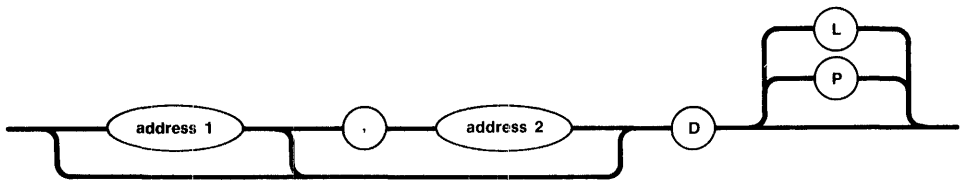
```
*20,21C

20:   IF change >= 50 THEN
21:       CALL next$money (50);
22:
*
```

You do not have to type the line numbers in the new text. EDIT automatically displays the numbers if you have entered the number command (N) during the editing session.

D — Delete

The Delete command (D) lets you erase or delete text. You can use this command to delete one line or many lines as shown in the following pictorial syntax.



MARKER POSITION

After executing this command, EDIT moves the marker to the line after the last line you deleted. If you deleted the last line of the file, EDIT moves the marker to the new last line.

PARAMETERS

no address	EDIT finds the current line and deletes it.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT deletes the addressed line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT deletes all text between the first address and the second address, inclusive.
L (list)	When you use a List option (L) with the Delete command , EDIT displays *all* characters, including nonprinting ASCII characters, which now occupy the line or lines you addressed.
P (print)	When you use a Print option (P) with the Delete command , EDIT displays all printing characters which now occupy the line or lines you addressed.

After you delete the text, EDIT automatically resequences the numbers.

Examples:

Suppose that you have the following piece of text in your file.

```
10:  DO WHILE i > 8;
11:      CALL next$coin (5);
12:  END;
```

Delete line 10 and print the text which now occupies line 10.

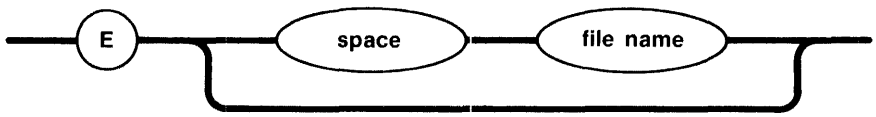
```
*10DP
```

EDIT responds by listing the CALL statement. Notice that the line numbers automatically shifted after you deleted the DO WHILE statement.

```
10:      CALL next$coin (5);
```

E — Edit

The Edit command (E) allows you to edit files already written on secondary storage. The Edit command also allows you to create a new file.



MARKER POSITION

If you are creating a new file, the marker will be at line zero. If the file already exists, the marker will be at the last line of the file.

PARAMETERS

space	You must type a space between this command and the file name.
file name	If you do not type a file name, EDIT uses the file name it currently remembers. You must have used this name previously, when you invoked EDIT or in one of the following commands: E, F, W, or R. If EDIT does not remember any file names, it will return an error message (?). If you type the name of a file that already exists, EDIT answers with the file name and the number of lines and bytes in the file. EDIT brings the file from secondary storage and copies it into the buffer. If you type a file name that does not exist, the EDIT program will respond with the file name followed by a colon and the words "new file."

Examples:

Suppose you are in the EDIT program and you wish to edit a file on a device called ":f1:" with the file name "prog/samp.plm". You would type

```
*E :f1:prog/samp.plm
```

EDIT responds with

```
:f1:prog/samp.plm: 23 lines, 857 bytes
*
```

This file happens to have 23 lines and 857 bytes. You can also use the Edit command to create new files. The following command creates a file called "prog/test.plm" on device 1. (Remember that the file will not be saved on secondary storage until you use the Write or Exit commands.)

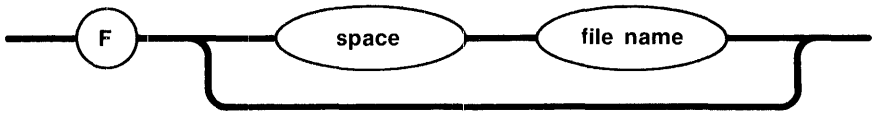
```
*E :f1:prog/test.plm
```

EDIT answers with

```
:f1:prog/test.plm: new file
*
```

F — File Name

You can use this command to display or change the remembered file name.



MARKER POSITION

This command does not move the marker.

PARAMETERS

space	You must type a space between this command and the file name.
file name	If you do not type a file name, EDIT responds with the file name it currently remembers. You must have used this name previously, when you invoked EDIT or in one of the following commands: E, F, W, or R. If EDIT does not remember any file names, it will display a blank line. If you type a file name different from the one EDIT remembers, the EDIT program changes the remembered name. This command does *not* rename a file if you have already written to secondary storage; it simply changes the name EDIT remembers.

Examples:

Suppose you are currently editing a file called "prog/change.p86". Change the remembered name to "prog/coin.p86."

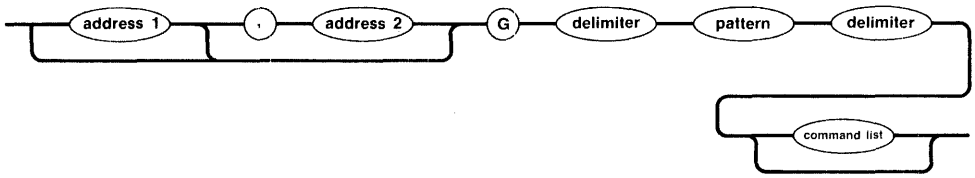
```
*F prog/coin.p86
```

EDIT responds by echoing the new file name.

```
prog/coin.p86
*
```

G — Global

The Global command directs EDIT to search the addressed text for every occurrence of the pattern you choose. It also tells EDIT to execute a command or several commands on each such line. The Global command sets the marker to each line addressed and executes the command list.



MARKER POSITION

After executing this command, EDIT moves the marker wherever the last command in the command list executed by "G" left it.

PARAMETERS

no address	EDIT assumes the address is the entire buffer (1,$), and it scans the text for every line that contains the pattern.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT either displays the addressed line or it executes the commands in the command list on this line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive.
pattern	EDIT looks through the addressed text and finds scans the text for every line that does not contain the pattern you typed between the delimiters.
command list	You can use any command with the Global command except another Global or a "V." You should enter these commands as shown in their pictorial syntax.

Examples:

Suppose your file contains the following lines of text.

```
END next$coin;
      CALL next$coin (50);
      CALL next$coin (25);
      CALL next$coin (10);
```

If you want to print every line that contains the word CALL you would type

```
*G/CALL/P
```

EDIT displays every line in the file which contains the word CALL.

```
CALL next$coin (50);
CALL next$coin (25);
CALL next$coin (10);
```

If you wanted to change every occurrence of COIN to MONEY you would use a Global command and specify Substitute and Print as the command list portion.

```
*G/coin/S//money/GP
```

This combination causes EDIT to replace "coin" with "money" throughout the entire buffer and print the changed lines.

```
END next$money;
      CALL next$money (50);
      CALL next$money (25);
      CALL next$money (10);
```

How Global Works

The Global command tries to be transparent in that it temporarily saves the command list exactly as you typed it. (The only exception is a global which contains one or more backslashes with carriage returns. This is explained in the section on interpreting commands in Chapter 6.) It then finds the first line which contains the string you specified in the global part of the command and executes the saved command list.

For example, suppose your file contains the following text.

```
ED SCHMIDT, 77 NW DRIVE, NEWBURG
JOHN SCHMIDT, 85 SW DRIVE, ORLANDO
SALLY SCHMIDT, 99 NE DRIVE, SPOKANE
SAMUEL SCHMIDT, 34 SE DRIVE, WOODVILLE
```

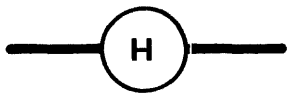
If you want to replace all of the commas with semicolons, you can type:

```
*G/,/S//;/G
```

EDIT actually takes the command list (the Substitute portion) and temporarily saves it. EDIT then executes the first part of the command by finding the first comma. It then retrieves the command list and executes it. The EDIT program repeats this process for every comma it finds.

H — Sets Tab Length

You can use the "H" command to change the spacing of tab settings. This is especially convenient if you need to write a program in different languages. For example an Assembler may expand tabs to eight spaces in the listing whereas a PL/M or PASCAL Compiler may use four spaces.



EDIT automatically uses four spaces for the tab settings. If you type an "H" you change this spacing to eight. To reverse the process and get your tab settings back to four spaces, type the "H" command again.

MARKER POSITION

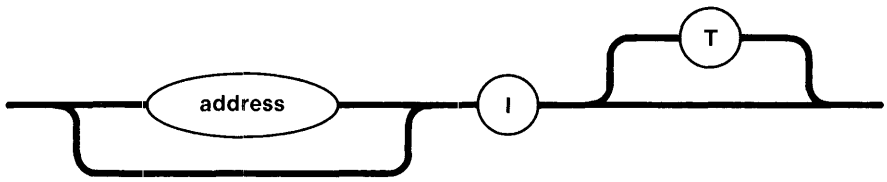
This command does not change the position of the marker.

PARAMETERS

The "H" command requires no parameters.

I — Insert

The Insert command (I) allows you to insert lines of text *before* the line you specify by your choice of address.



MARKER POSITION

After executing this command, EDIT moves the marker to the last line you inserted.

PARAMETERS

no address	EDIT finds the current line and allows you to insert text before it.
address	EDIT finds the address you choose and allows you to insert text before it.
T (tab)	When you use a Tab option (T) with the Insert command, EDIT automatically begins your next line at the same tab setting you chose for the previous line. If you want to add another tab, simply press the TAB key. If you want to begin your next line at a lesser tab setting, type two periods (..) followed by a carriage return on an otherwise blank line. This will set the tab back one from the previous line. EDIT will not include the periods in the file, whereas the tabs become part of the file.

You choose the location of the new text by means of the previously described parameters. After you have added all the text you wish, type a period (.) and a carriage return on an otherwise blank line to stop inserting.

Examples:

Suppose your file contains the following lines of text.

```
21:   END next$coin;
22:           CALL next$coin (50);
23:           CALL next$coin (25);
24:           CALL next$coin (10);
```

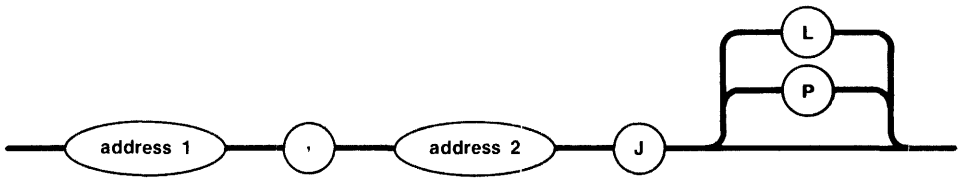
Insert a CALL (75) statement before line 22.

```
*22I
22:           CALL next$coin (75);
23:
```

You do not have to type the line numbers in the new text. EDIT automatically writes the numbers if you have used the number command (N) during the editing session.

J — Join

You can use the Join command (J) to merge two or more lines which you specify by address.



MARKER POSITION

After executing this command, EDIT moves the marker to the line you just constructed with the Join command.

PARAMETERS

address 1	EDIT finds the first address you specify.
comma (,)	The comma separates the first address from the second address.
address 2	After you chose a first address, EDIT joins all text between the first address and the second address, inclusive.
L (list)	When you use a List option (L) with the Join command, EDIT displays *all* characters, including nonprinting ASCII characters, in the line you constructed.
P (print)	When you use a Print option (P) with the Join command, EDIT displays all printing characters in the line you constructed.

Example:

Suppose your file contains the following lines.

```
12:             change = change - x;
13:  END
14:  $next$money;
```

Join lines 13 and 14 and add the Print option so you can verify your changes.

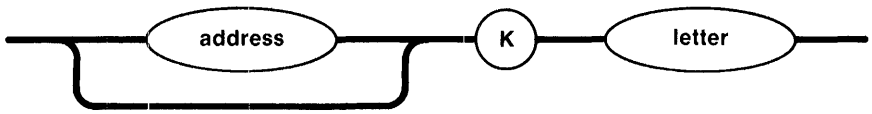
```
*13,14JP
```

The corrected line looks like this

```
13:  END$next$money;
```

K — Marks A Line

The "K" command gives you another way of marking or referring to a line. You can specify a line and mark it with any letter of the alphabet. Once you specify a letter, as shown in the following syntax, you can find the line by typing a single-quote mark (') and the letter.



Once you have marked a line, this mark is associated with the line even if EDIT resequences the line numbers. However, if you ever delete or substitute anything *within* a "marked" line, EDIT also deletes the "mark."

MARKER POSITION

The "K" command does not move the marker.

PARAMETERS

no address	EDIT finds the current line.
address	EDIT finds the address you specify.
letter	EDIT marks the line you addressed with the letter you choose.

Examples:

Suppose your file is quite large but within it there is a line that you have to refer to repeatedly. You could mark it with the "K" command and refer to it whenever necessary.

For example, suppose the following lines are part of a large file. If you had to go back to statement 8 often, you could save time by marking it with the "K" command

```
8:    PROCEDURE (X);
9:              MONEY (I);
```

If you type

```
*8Ka
```

you can refer to this line with a single quote (') and the letter "a".

By typing

```
*'a
```

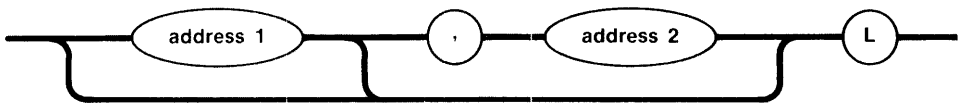
you position the pointer at the PROCEDURE statement you marked with the "K" command.

EDIT responds with

```
8:              PROCEDURE (X);
```

L — List Nonprinting ASCII Characters

The List command (L) displays *all* characters, including nonprinting ASCII characters, between the lines you specify by address. The command is similar to Print except that it also displays nonprinting characters, such as tabs. Some of these characters appear as representations (such as horizontal arrows for tabs) and some appear as hexadecimal numbers preceded by a backslash (\). For example, a control-E prints as "\05".



MARKER POSITION

After executing this command, EDIT moves the marker to the last line you listed.

PARAMETERS

no address	EDIT finds the current line and displays it.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT displays the addressed line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT displays all text between the first address and the second address, inclusive.

Examples:

Suppose your file contains the following lines.

```
12:              change = change - X;
13:   END;
14:   next$money;
```

You need to know if there are any nonprinting ASCII characters which could be causing errors to occur when you compile the program. Enter:

```
*12,14L
```

Suppose EDIT then replies as follows.

```
12:              change = change - X;
13:   END
14:   next$mone\05y;
```

You can see that you do have some undesirable ASCII characters. You can find out what the numbers mean by looking them up on the ASCII to HEX reference table in Appendix A. Delete the ASCII character using the period to represent the "\05" in a substitute command.

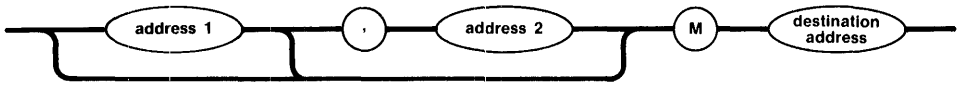
```
14S/mone.y/money/L
```

EDIT will answer with the corrected line.

```
14:   next$money;
```

M — Move

This command (M) allows you to move one or more lines of text from one place in a file to another. You can specify which lines you want moved and where you want them placed by your choice of parameters.



MARKER POSITION

After executing this command, EDIT moves the marker to the new position of the last line you moved.

PARAMETERS

no address	EDIT finds the current line and moves it after the line specified in the destination address.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT moves the line addressed after the line specified by the destination address.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive. It moves this text so that it follows the line specified by the destination address.
destination address	EDIT finds the destination address you specify and moves the text identified by the second and/or first addresses after this line.

Examples:

Suppose your file contains the following lines of text.

```
21:  END next$coin;
22:          CALL next$coin (50);
23:          CALL next$coin (25);
24:          CALL next$coin (10);
```

You can use the Move command to place line 21 after the original line 24

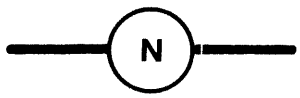
```
*21M24
```

The text should read as follows.

```
21:          CALL next$coin (50);
22:          CALL next$coin (25);
23:          CALL next$coin (10);
24:  END next$coin;
```

N — Number

This command (N) tells EDIT to display the line numbers asociated with the buffer or file being edited. If you have already used the Numbering command (N) and you type it again, EDIT will stop displaying the line numbers.



You can use the Numbering command immediately after you invoke the EDIT program. This causes everything you append, alter, or create to have a visible line number associated with it. You can also invoke the Numbering command if you have been editing the file or buffer without numbers. These numbers are an EDIT feature and they are not saved when a you write a file onto secondary storage.

MARKER POSITION

This command does not change the marker position.

PARAMETERS

The Numbering command has no parameters.

Examples:

Suppose your file consists of the following lines.

```
make$change:
    DO;
            DECLARE money(8) BYTE;
            DECLARE change BYTE;
            DECLARE change BYTE;
```

You can number the lines by typing this command.

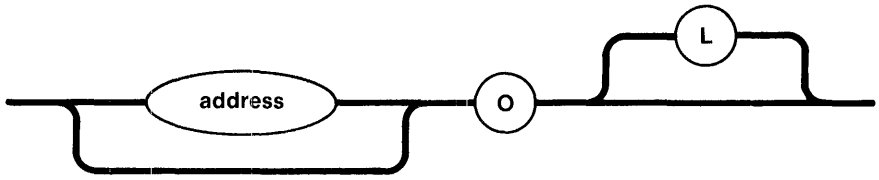
```
*N
```

If you print the buffer, you can see that the text now has visible line numbers to the far left.

```
1:  make$change:
2:  DO;
3:          DECLARE money(8) BYTE;
4:          DECLARE change BYTE;
5:          DECLARE change BYTE;
```

O — Over One Screen

The "O" command displays the current line of the file plus up to 22 more lines. This command is equivalent to the Print command ".,+22 P". Therefore it saves you keystrokes. As with the Print command, EDIT displays the lines in their original order.



MARKER POSITION

This command moves the marker to the last line displayed.

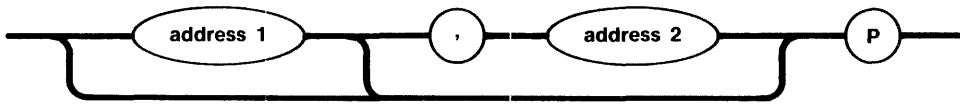
PARAMETERS

no address	EDIT finds the current line and displays it plus up to 22 additional lines.
address	EDIT finds the address you choose and displays that line plus up to 22 additional lines.
L (list)	When you use a List option (L) with the "O" command, EDIT displays *all* characters, including nonprinting ASCII characters, in the lines that it displays.

This command is more useful on a screen-type terminal than a hard-copy or single-line terminal, since it displays the one screen full of text starting with the line you specify.

P — Print

You can use the Print command (P) to display the text between the lines you specify by address.



MARKER POSITION

After executing this command, EDIT moves the marker to the last line you printed.

PARAMETERS

no address	EDIT finds the current line and displays it.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT displays the addressed line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT displays all text between the first address and the second address, inclusive.

Examples:

Suppose the following lines of text make up your file.

```
1:    make$change:
2:        DO;
3:              DECLARE money(8) BYTE;
4:              DECLARE change BYTE;
5:              DECLARE change BYTE;
6:
7:              next$money:
8:                  DECLARE X BYTE;
9:                  money(I) = X;
10:                 PROCEDURE(X);
```

Print the first 6 lines.

```
*1,6P
```

EDIT responds with

```
1:    make$change:
2:        DO;
3:              DECLARE money(8) BYTE;
4:              DECLARE change BYTE;
5:              DECLARE change BYTE;
6:
*
```

Now, suppose that this same file has no numbers associated with it and you want to print the entire buffer.

```
*1,$P
```

EDIT responds with

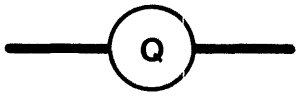
```
make$change:
    DO;
                DECLARE money(8) BYTE;
                DECLARE change BYTE;
                DECLARE change BYTE;

                next$money:
                    DECLARE X BYTE;
                    money(I) = X;
                    PROCEDURE(X);

*
```

Q — Quit

The Quit command (Q) lets you leave the EDIT program.



This command quits the EDIT program and does not write the buffer to secondary storage (see the Write command). This is especially useful if you have done some careless editing and want to start over.

R — Read

The Read command (R) allows you to add the contents of a file *after* a line in your buffer.

MARKER POSITION

After executing this command, EDIT moves the marker to the last line it read into the buffer.

PARAMETERS

no address	When you do not specify an address, EDIT places the contents of the file you specify after the last line of the text.
address	EDIT places the contents of the file you specify after the line you specify with this address. Remember, you *can* use zero as an address with the Append command.
space	You must type a space between this command and the file name.
file name	This is the name of the file from which EDIT reads. EDIT finds the file you specify by a file name and places the entire contents after the address you choose. If you do not type a file name, EDIT responds with the file name it currently remembers. You must have used this name previously, when you invoked EDIT or in one of the following commands: E, F, W, or R,. If EDIT does not remember any file names, it will return an error message (?).

Examples:

Suppose the following lines compose a file called "samp.txt".

```
DO WHILE I < 8;
    CALL next$coin (5);
END;
```

Your buffer contains the following lines.

```
DO;
    DECLARE money(8) BYTE;
    DECLARE change BYTE;
    DECLARE I BYTE;
```

Use the Read command (R) to add "samp.txt" to the buffer.

```
*R samp.txt
```

EDIT responds with

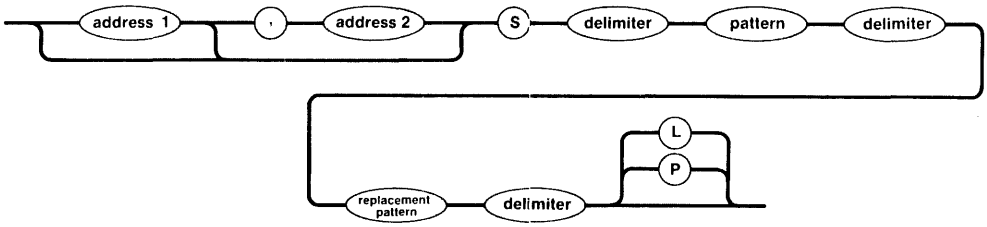
```
samp.txt: 3 lines, 43 bytes.
```

Your buffer now contains the following text.

```
DO;
    DECLARE money(8) BYTE;
    DECLARE change BYTE;
    DECLARE I BYTE;
DO WHILE I > 8;
    CALL next$coin (5);
END;
```

S — Substitute

The Substitute command (S) allows you to replace the occurrences of a character, word or phrase in the line or lines which you specify by address.



MARKER POSITIION

After you execute this command, the marker will be at the last line in which you substituted for a pattern.

PARAMETERS

no address	EDIT finds the current line.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT substitutes the replacement pattern for the pattern in the addressed line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive.
delimiter	EDIT reads the first character after the "S" as the delimiter. EDIT considers every character after the delimiter as part of the pattern until it finds another instance of the delimiter.
pattern	This is the character, word, or phrase you want to replace. You must type it between delimiters.
replacement-pattern	This is the character, word, or phrase with which you replace the first pattern. You must type it between delimiters.
G (Global)	When you use the Global option (G) with the substitute command, EDIT substitutes for *all* cases of the pattern in the lines you addressed. If you do not use the Global option, EDIT substitutes only for the first case of the pattern.
L(list)	When you use a List option (L) with the Substitute command , EDIT displays *all* characters, including nonprinting ASCII characters, in the last line EDIT actually substituted for the pattern.
P (print)	When you use a Print option (P) with the Substitute command , EDIT displays all printing characters in the last line EDIT actually substituted for the pattern.

S — Substitute (Continued)

Examples:

Suppose your file consists of the following lines of text.

```
1:    make$money:
2:    DO;
3:          DECLARE money(8) BYTE;
4:          DECLARE change BYTE;
```

Replace the first occurrence of "money" with "coin" in lines 1 through 3 using the Substitute command and Print the results.

```
*1,3S/money/coin/P
```

EDIT responds with the last line it changed.

```
3:          DECLARE coin(8) BYTE;
```

Suppose your file contains the following lines.

```
DECLARE coin(8) BYTE;          /*this is the result*/
DECLARE I BYTE;                /*index to coin array*/
next$coin;                     /*this is a procedure//
```

The line beginning with NEXT$COIN has a mistake in the comments portion. The second slash should be an asterisk (*). You can use the Substitute command to fix this, but if you use the slash as a delimiter, you will have problems. You can correct this line easily by using the another character as a delimiter. For example

```
*/next/S1procedure/1procedure*1P
```

corrects the problem and treats the slash as a normal character. The Print option (P) directs EDIT to display the line.

```
next$coin;                     /*this is a procedure*/
```

Now, suppose that you want to delete the word BYTE from line one of the previous file, but you don't want to delete the whole line. You can do this by substituting a "nothing" or no character in place of BYTE.

```
*1S/BYTE//
```

Special Case

You can rearrange the text within lines by using "\(" and "\)" as "tags" for the parts you want to move. Suppose you have a file consisting of the following names.

```
Johnson, Edward
Johnson, Edwin
Johnson, Elisa
Jonas, Jessie
Jonas, Robert
```

Now suppose that you want the last name to follow the first with no commas. You can do this with a series of EDIT commands but it can be tedious. However, you can "tag" the pieces of the pattern and rearrange these pieces. EDIT remembers the part of the pattern enclosed in the "\(" and the "\)" on the left side of the Substitute command so this part can be used on the right side. On the right side, the symbol "1" refers to whatever matches the first pair of "\(...\)". The symbol "2" refers to the second pair of "\(...\), and so on.

S — Substitute (Continued)

Type the following command.

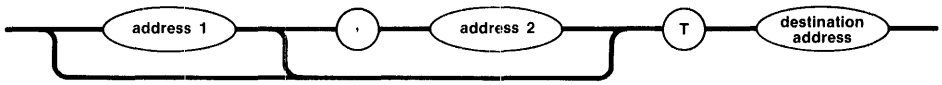
```
*1,$S/!\([!,]*\), *\(.*\)/\2\1/
```

Although this command is hard to read, it rearranges the text within the lines without typing many different EDIT commands. The first pair of "\(...\)" matches any string up to the comma (the last name). The second pair of "\(...\)" matches whatever follows the comma (the first name). So, the file now reads as follows.

```
Edward Johnson
Edwin Johnson
Elisa Johnson
Jessie Jonas
Robert Jonas
```

T — Text Copy

The Text Copy command (T) allows you to copy one or more lines of text. You can specify which lines you want copied and where you want them placed by your choice of address. This command is similar to the Move command (M) except EDIT copies the lines rather than moving them.



MARKER POSITION

After executing this command, EDIT moves the marker to the new position of the last line you copied.

PARAMETERS

no address	EDIT finds the current line and copies it after the line specified in the destination address.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT copies the line addressed after the line specified by the destination address.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive. It copies this text after the line specified by the destination address.
destination address	EDIT copies the text identified by address and copies this text so that it follows the line specified by the destination address.

Examples:

Suppose your file contains the following lines.

```
8:              change = change - X;
9:    END
10:   $next$money;
```

You can use the Text Copy command (T) to copy the "change" line and place it after "next$money".

```
*8T10
```

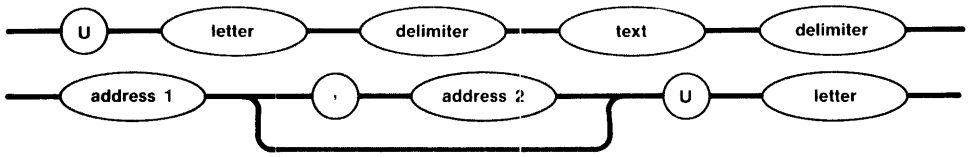
If you print these lines they should read as follows.

```
8:              change = change - X;
9:    END
10:   $next$money;
11:             change = change - X;
```

U — User Macro

A macro is a set of statements to which you assign a name by means of the macro or "U" command. You can use the user-macro command (U) to define a macro, list the defined macro, and display the definition of a given macro. Refer to Chapter 6 for examples and a more detailed explanation of macros.

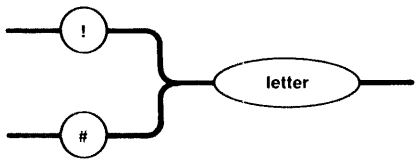
Defining a Macro



Displaying the Defined Macro

Listing the Defined Macro

Invoking a Macro



MARKER POSITON

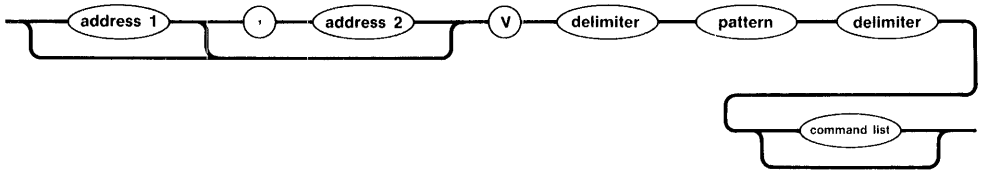
The User Macro (U) command does not move the marker.

PARAMETERS

address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT defines the macro to be this line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive. It then defines the macro to be this text.
letter	EDIT specifies the macro by the letter you choose.
delimiter	EDIT reads the first character after the letter as the delimiter. EDIT considers every character after the delimiter as part of the text until it finds anther instance of the delimiter.
text	The text you enter becomes the contents of the Macro.
pound sign (#)	The pound sign (#) and the letter you chose, invokes the macro specified by the letter.
exclamation point (!)	The exclamation point (!) and the letter you chose invokes the macro. However, it causes EDIT to ignore any nested macro invocations.

V — Exclusive Global

The "V" command directs EDIT to search the addressed text for every occurrence of a line that does *not* contain the pattern you choose. This command is like the Global command except that it uses lines that don't match.



MARKER POSITION

After executing this command, EDIT moves the marker wherever the last command in the command list executed by "V" left it.

PARAMETERS

no address	EDIT assumes the address is the entire buffer, (1,$) and it scans the text for every line that does not contain the pattern.
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT either displays the line or performs the commands in the "command list" on the addressed line.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive and scans it for every line that does not contain the pattern you specify.
pattern	EDIT looks through the addressed text and finds scans the text for every line that does not contain the pattern you typed between the delimiters.
command list	You can use any command with the Exclusive Global command (V) except another Exclusive Global or a "G." You should enter these commands as shown in their pictorial syntax.

Examples:

Suppose your file contains the following lines of text.

```
END next$coin;
      CALL next$coin (50);
      CALL next$coin (25);
      CALL next$coin (10);
```

If you want to print every line that does not contain the word CALL you would use the "V" command.

```
*V/CALL/P
```

EDIT displays every line in the file which *does not* contain the word CALL.

```
END next$coin;
*
```

W — Write

The Write command (W) allows you to write part or all of the contents of the buffer into a file on secondary storage.

You can write pieces of the text into a file as well as the entire buffer. If the file already exists, it is replaced by the contents of the addressed lines.

MARKER POSITION

The Write command does not change the position of the marker.

PARAMETERS

no address	EDIT assumes no address to be the contents of the entire buffer (1,$).
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT writes the addressed line onto secondary storage.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive.
space	You must type a space between this command and the file name.
file name	If you do not type a file name, EDIT responds with the file name it currently remembers. You must have used this name previously, when you invoked EDIT or in one of the following commands: E, F, W, or R. If EDIT does not remember any file names, it will return an error message (?). If you type a file name, EDIT will write the contents of the buffer (or the addressed part) into secondary storage. It will then display the file name followed by the number of lines and the number of bytes in the file. If you type the name of a file that already exists, EDIT writes the contents of the buffer over the existing file thus destroying any contents previously contained in the file.

Examples:

Suppose you want to save the following lines of text.

```
DO WHILE 18;
    CALL next$coin (5);
END;
```

W — Write (Continued)

Write this text into a file on device 1 using the name "samp.txt".

```
*W :f1:samp.txt
```

The EDIT program answers with

```
:f1:samp.txt: 3 lines, 43 bytes.
*
```

Now,suppose your buffer consists of the following lines.

```
1:    make$change:
2:    DO;
3:            DECLARE money(8) BYTE;
4:            DECLARE change BYTE;
5:
6:            next$money:
7:                DECLARE X BYTE;
8:                money(I) = X;
9:                PROCEDURE(X);
```

If you want to save only lines 1 through 5, you can write them out to secondary storage.

```
*1,5W :f1:change.plm
```

EDIT answers with

```
:f1:change.plm: 5 lines, 69 bytes.
```

X — Exit

The Exit command (X) is equivalent to the Write commmand followed by the Quit command. This command writes the contents of the buffer into the file you specify or into the last file it remembers. It then quits the EDIT program.

MARKER POSITION

The Exit command does not change the position of the marker.

PARAMETERS

no address	EDIT assumes no address to be the contents of the entire buffer (1,$).
address 1	EDIT finds the first address you specify. If this is the only address you choose, EDIT writes the addressed line onto secondary storage.
comma (,)	The comma separates the first address from the second address.
address 2	EDIT finds all text between the first address and the second address, inclusive.
space	You must type a space between this command and the file name.
file name	If you do not type a file name, EDIT responds with the file name it currently remembers. You must have used this name previously, when you invoked EDIT or in one of the following commands: E, F, W, or R. If EDIT does not remember any file names, it will return an error message (?). If you type a file name, EDIT will write the contents of the buffer (or the addressed part) into secondary storage and display your file name along with the number of lines and bytes. It will then exit the EDIT program and your system will display its prompt. If you type the name of a file that already exists, EDIT writes the contents of the buffer over the existing file thus destroying any contents previously contained in the file.

Examples:

Suppose your buffer consists of the following lines.

```
1:  make$change:
2:  DO;
3:          DECLARE money(8) BYTE;
4:          DECLARE change BYTE;
5:
6:          next$money:
7:              DECLARE X BYTE;
8:              money(I) = X;
9:              PROCEDURE(X);
```

X — Exit (Continued)

Write the contents of the buffer out to secondary storage and quit EDIT at the same time.

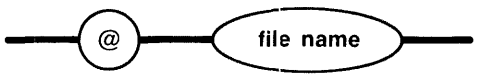
```
*X :f1:prog/text.plm
```

EDIT responds with the following line and the prompt for your system.

```
:f1:test.plm: 10 lines, 138 bytes.
```

@ — Command File

A command file is a series of EDIT commands to which you assign a name just as you would any other file. The EDIT program executes this file as a series of commands when you enter an "at" sign (@) followed by a file name. Refer to Chapter 6 for examples and a more detailed explanation of Command Files.



MARKER POSITION

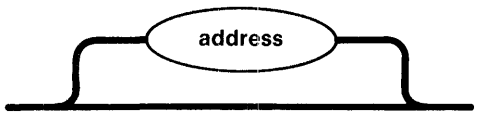
The Command File command does not change the position of the marker.

PARAMETERS

file name	After you type a file name, EDIT will execute the contents of the file as a series of EDIT commands.

C/R — Displaying A Specific Line

EDIT finds the addressed line and displays it. If you do not address a line (just type a carriage return), EDIT displays the next line. This is convenient if you want to look at the file a line at a time.



MARKER POSITION

After executing this command, EDIT moves the marker to the last line you displayed.

PARAMETERS

no address	If you type a carriage return on an otherwise blank line, EDIT displays the line after the current line. If the marker is at the last line of the buffer, EDIT displays nothing.
address	EDIT displays addressed the line.

Examples:

Suppose your file consists of the following text and your marker is on line 1.

```
1:   make$change:
2:   DO;
3:           DECLARE money(8) BYTE;
4:           DECLARE change BYTE;
```

If you want to examine the text line by line, type a carriage return on an otherwise blank line.

```
*(c/r)
```

EDIT responds as follows.

```
2:   DO;
```

If you type another carriage return, EDIT answers with the next line.

```
*(c/r)
3:           DECLARE money(8) BYTE;
```

Now suppose that you want to see line 1. Type:

```
*1
```

EDIT answers with

```
1:   make$change:
```

* — Comment

The Comment command (*) tells EDIT to ignore the text which follows the asterisk. This command allows you to add comments to your edit session which may make it more readable.

MARKER POSITION

The Comment command (*) does not change the position of the marker.

PARAMETERS

text	The text you enter is the comment.

Example:

This command is especially useful in Command Files (see Chapter 66) since EDIT reads them as a series of EDIT commands. Suppose your command file consists of the following text.

```
G/BYTE/S//ADDRESS/
G/change/D
$I
END;
.
```

You need to insert some comments so you don't forget what the file is supposed to do.

```
*1I
*This Command File replaces BYTE with ADDRESS and deletes all lines
*containing the word CHANGE. It also inserts the word END and a ";" before
*the last line of the file.
*
.
```

The file now contains the following lines.

```
*This Command File replaces BYTE with ADDRESS and deletes all lines
*containing the word CHANGE. It also inserts the word END and a ";"
*before the last line of the file.
*
G/BYTE/S//ADDRESS/P
G/change/D
$I
END;
.
```

CHAPTER 6
ADVANCED EDITING

This chapter describes how to use more powerful EDIT features such as command files and macros.

- A *command file* is a series of *EDIT commands* to which you assign a name just as you would any other file. The EDIT program executes this file as a series of commands when you enter an "at" sign (@) followed by a file name.
- A *macro* is text to which you assign a name by means of the macro or "U" command. The EDIT program inserts the contents of a macro wherever you type a pound sign (#) or an exclamation point (!) and the name of the macro.

Command Files

Command files are files you create especially for use in EDIT. You create them just as you would any other file. However, these files contain commands which the EDIT program automatically executes when you type an "@" and the file name during an editing session.

The general form for this editing technique is as follows.

You typically use command files for long command sequences that you execute often. Rather than entering a long series of EDIT commands, you call the predefined command file with a single command line. This facility speeds your work and reduces the typing errors usually associated with a long string of commands.

Examples:

Suppose you want to edit many files. Each file contains the word BYTE in numerous places and you must change it to ADDRESS and print the results. There are also numerous references to "change" which you must delete. And finally, you need to add "END;" before the last line of these files.

First, you need to create a command file. This manual will refer to the command file as ":f1:assist/aid.cmd" where ":f1:" is the logical name for the device and "assist/aid.cmd" is the file name. Enter EDIT and create the commands as if you were executing them on a line.

Type

```
ED :f1:assist/aid.cmd
```

EDIT responds with a sign-on (not shown) and an acknowledgment of your file name.

```
:f1:assist/aid.cmd: new file
*
```

In order to actually create this file you must append the commands you want in your command file.

```
*a
G/BYTE/S//ADDRESS/GP          (changes BYTE to ADDRESS and print)
G/change/D                    (deletes lines containing change)
$I                            (inserts END; before
END;                          the last line)
x                             (holds place for period)
.
```

EDIT answers with a prompt.

```
*
```

Before you write the text into the file, notice that the insert command (I) does not have a period after the END line to tell EDIT to stop inserting. Instead of a period, you typed an "x". If you had typed a period, EDIT would have stopped appending any commands into the file because it would have read the period as a signal to stop adding to the file. The letter "x" is holding a place for the period. Just use the substitute command to replace the "x" with a period (.).

```
5S/x/./
```

Now, write the file into secondary storage. EDIT will remember the file name you previously specified.

```
*W
```

EDIT responds

```
:f1:assist/aid.cmd: 5 lines, 46 bytes.
*
```

Your command file is now on secondary storage.

Suppose "program/coin.plm" is one of the files you need to alter. If you did the examples in the tutorial, you already have this file in secondary storage. If you have not gone through the tutorial you can enter the following text and write it into a file called "program/coin.plm."

```
make$change:
    DO;
            DECLARE coin(8) BYTE;           /*this is the result*/
            DECLARE change BYTE;            /*number to be converted*/
            DECLARE I BYTE;                 /*index to coin array*/

            next$coin:                      /*this is a procedure*/
              PROCEDURE(X);
                DECLARE X BYTE;             /*X is specified*/
                coin(I) = X;
                I = I + 1;
                change = change - X;
            END next$coin;

            change = 100 -    ;             /*write the cost here*/
            I = 0;                          /*initialize the index*/
            DO WHILE change >= 50;          /*half dollar*/
              CALL next$coin(50);
            END;
            DO WHILE change >= 25;          /*quarters*/
              CALL next$coin(25);
            END;
            DO WHILE change >= 10;          /*dimes*/
              CALL next$coin(10);
            END;
            DO WHILE change >= 15;          /$nickels*/
              CALL next$coin(5)
            END;
            DO WHILE change >= 1;           /*pennies/*
              CALL next$coin(1);
            END;
            DO WHILE change >= 8;
              CALL next$coin(0)             /*zero out rest of coins*/
            END;
    END make$change;
```

Enter the file through EDIT and use the command file to make the necessary changes.

```
*E program/coin.plm
```

EDIT answers with

```
program/coin.plm: 35 lines, 929 bytes.
*
```

Now that you have entered the program that you want to alter, you can invoke the command file by typing.

```
*@ :f1:assist/aid.cmd
```

EDIT executes the commands contained in ":f1:assist/aid.cmd" on the file "program/coin.plm". The following display is a result of the command file's first command which directed EDIT to print the corrected lines containing ADDRESS.

```
      DECLARE coin(8) ADDRESS;          /*this is the result*/
      DECLARE change ADDRESS;           /*number to be converted*/
      DECLARE I ADDRESS;                /*index to coin array*/
        DECLARE X ADDRESS;              /*X is specified*/
*
```

The command file will remain on secondary storage after you exit EDIT.

Command Files Within Command Files

You can call command files within command files. EDIT finds the first command file and executes the contents up to the call for the second command file. It then finds the second command file and executes the contents and so on up to nine calls. As EDIT exhaust each file, it returns to the previous one.

The Macro Feature

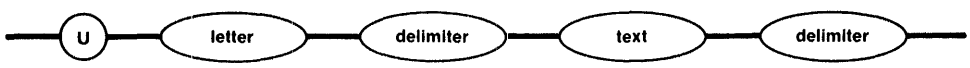
EDIT macros, like command files, are another time saving feature. However, macros are *not* necessarily a series of EDIT commands. They can be used in a number of different ways. A macro can be a group of lines that you refer to by a single name or a procedure you use often. Macros are stored in a temporary memory and EDIT expands them when you call them by name.

Defining A Macro

You can use the user macro command (U) to define a macro, to list the defined macros, and to display the definition of a given macro.

To define a macro, you type the user macro command (U) followed by a letter of the alphabet that will serve as the name of the macro and the text of the macro enclosed in slashes (//). The slashes are delimiters. EDIT reads the first character after the "letter" as the delimiter. You can use any character you wish as a delimiter, but to avoid confusion this manual will use a slash.

The general form for defining a macro is



Suppose you want to create a macro of lines you may use often in writing programs. If you have to use the same DECLARE statements for several programs, you may find it useful to put these statements in a macro rather than typing them each time. Be sure to use a backslash before the carriage return so EDIT does not try to define an incomplete macro.

```
*Uf/DECLARE X BYTE;\
DECLARE Y ADDRESS;\
DECLARE Z BYTE;/
```

Now, you have defined "f" as a macro. In order to display the contents of this macro, type

```
*uf
```

EDIT answers with

```
DECLARE X BYTE;
DECLARE Y ADDRESS;
DECLARE Z BYTE;
```

The general form for using a macro is

Suppose you are creating a file that requires the DECLARE statements you just wrote into the macro "f". To enter the lines contained in the previous macro, type a pound sign (#) and the letter "f" which represents the contents. Type the following example.

```
*A
making$change:
DO;
#f                                  (this line writes the contents of macro "f" here)
NEXT$coin;
    PROCEDURE (X);
```

EDIT copies the text associated with the macro into the buffer where you typed the pound sign (#) and the letter "f." However, EDIT does not display the contents of the macro immediately; you must request a display of the text by using another EDIT command.

```
*1,$P
```

EDIT answers with the complete file.

```
making$change:
DO;
DECLARE X BYTE;
DECLARE Y ADDRESS;
DECLARE Z BYTE;
next$coin;
  PROCEDURE (X);
```

The display shows that the DECLARE statements are included in the buffer. This macro can be used repeatedly during the editing sessions as long as you do not Exit or Quit the EDIT program. The macro just described is not saved after you leave EDIT because it is not written on secondary storage.

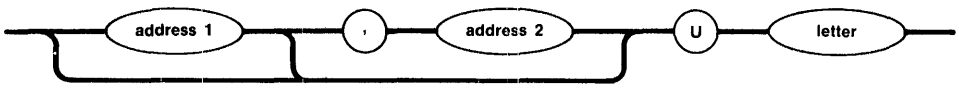
Line-Range Macros

The user macro command (U) allows you to define macros from the text in the buffer. For example, suppose your text consisted of the following lines:

```
1:    make$change:
2:        DO;
3:            DECLARE money(8) BYTE;          /*this is the result*/
4:            DECLARE change BYTE;            /*number to beconverted*/
5:            DECLARE I BYTE;                 /*index to money array*/
6:
7:            next$money:                     /*this is a procedure*/
8:                PROCEDURE(X);
9:                    DECLARE X BYTE;         /*X is specified*/
10:                       money(I) = X;
11:                       I = I + 1;
12:                       change = change - X;
```

Let's say that you know that you will need to use statements 11 and 12 often while writing this program. If you define a macro to be lines 11 and 12 of the buffer, you can save yourself typing time.

The general form for a line-range macro is



So, to define lines 11 and 12 as a macro you would type

```
*11,12Ua
```

You can now use this macro any time you need to write the statements in lines 11 and 12. You can use the macro by typing a pound sign (#) and the letter "a".

```
*#a
```

Macros And Command Files

You can define a macro to be a command file. This saves you even more time and reduces the chance of error.

To define a macro to be a command file, you must first create and write the file into secondary storage. Then you assign the invocation of the command file to a macro just as you would a normal macro. Suppose you want to use the same command file you created earlier. Since you have already written the file into secondary storage, you can assign it to a macro and substituting the command file invocation for the text.

```
*Uc;@:f1:assist/aid.cmd;
```

Notice that the delimiters are semicolons because a slash is used in the file name.

The previous EDIT statement defines the macro "c" to be the command file :f1:assist/aid.cmd. Now, rather than typing

```
*@:f1:assist/aid.cmd
```

you can type

```
*#c
```

and EDIT will execute the command file.

This macro can be used repeatedly during the editing sessions as long as you do not *exit* or *quit* the EDIT program. When you exit the EDIT program, the macro assignment is not saved because it is not written on secondary storage.

ED.MAC

You already know how to create, display, and execute macros and command files. But, up to this time, EDIT lost macro definitions when you left the EDIT program. ED.MAC is a special command file that EDIT executes automatically when you invoke the program.

NOTE

EDIT actually scans and executes the commands in ED.MAC before it opens the file you specify. For this reason, you must be careful when you use text-dependent commands such as "1,$P". If ED.MAC contains a text-dependent command before ED.MAC places any text in the buffer, the EDIT program will respond with a question mark (?). EDIT then aborts ED.MAC and does not load the file you specified.

You create this file just as you would any other command file. However, you must not use a prefix which designates a secondary storage device. This causes your system to put ED.MAC wherever it normally stores files when you do not specify a device. For example, iRMX 86 stores ED.MAC on the default directory.

EDIT is programmed to recognize the name ED.MAC so that you can create a set of commands which EDIT executes at the time it signs on and before you see a prompt. You can also define macros in ED.MAC so that you do not have to define them each time you invoke the EDIT program.

Examples:

Suppose you are ready to create an ED.MAC file. Think of some commands you have to execute each time you enter EDIT. You can include these commands in ED.MAC so that each time you invoke the EDIT program these commands are executed automatically. The numbering command is a good example of this type of command.

Place the numbering command (N) in ED.MAC and write it out to secondary storage.

```
*E ED.MAC
ED.MAC: new file
*A
N
.

*W
ED.MAC: 1 lines, 3 bytes.
```

Now, each time you invoke EDIT, the lines of the text you are altering or creating will be numbered and you don't have to type anything.

You can use the ED.MAC feature to define macros that you use often. Try placing the "f" macro you previously created into ED.MAC.

```
*E ED.MAC
ED.MAC: 1 lines, 3 bytes.
*A
2:    U1/DECLARE X BYTE;
3:    DECLARE Y ADDRESS;
4:    DECLARE Z BYTE;/
5:    .
*
```

Now, write the macro onto ED.MAC.

```
*W
```

EDIT answers as follows.

```
ED.MAC: 4 lines, 63 bytes.
```

EDIT executes this macro only if you call it by entering a pound sign (#) and the letter "f" even though ED.MAC saves the macro.

Macros Within Macros

You can invoke macros within another macro up to nine times. For example, suppose you define the macro "g" to be the following lines.

```
*Ug/make$change:
DO;/
```

Now, suppose that after you have written macro "g", you decide to include the contents of macro "g" in macro "y." If you don't want to rewrite macro "g", you can call it within macro "y." You must use a backslash before the pound sign (#) so EDIT won't try to expand the macro "g" while you are defining the macro "y."

```
*Uy/\ #g
DECLARE coin(8) BYTE;
DECLARE change BYTE;
DECLARE I BYTE;/
```

Suppose that the following lines make up your text.

```
1:    next$coin:
2:    PROCEDURE(X);
3:    DECLARE X BYTE;
4:    coin(I) = X;
```

Insert the macro "y" before line 1

```
*1I
1:    #y
6:    .
*
```

You can tell by the previous line numbers that EDIT has expanded the macros but you will have to display the buffer to make this expansion visible.

```
*1,$P
1:    make$change:
2:    DO;
3:    DECLARE coin(8) BYTE;
4:    DECLARE change BYTE;
5:    DECLARE I BYTE;
6:    next$coin:
7:    PROCEDURE(X);
8:    DECLARE X BYTE;
9:    coin(I) = X;
```

Disregarding Macros Within A Macro

The exclamation point (!) followed by a letter referencing a macro tells EDIT to disregard any macro invocation within the macro. It also automatically takes away any meaning from a carriage return at the end of a line until EDIT find another delimiter. You can use the exclamation point and the letter defined to represent the macro in the same way you would pound sign (#) and the letter.

Suppose you are editing a file similiar to that in the preceding example. However, this file already contains the contents of macro "g".

```
1:    making$change:
2:    DO;
3:    next$coin:
4:    PROCEDURE(X);
5:    DECLARE X BYTE;
6:    coin(I) = X;
```

Since the file already contains the contents of macro "g", you don't want to have it entered again. But you do want the rest of the contents of macro "y" expanded. The best thing to do is to insert the macro "y" and have EDIT disregard the call for macro "g".

```
*3I
3:    !y
6:    .
*
```

When you display the text, EDIT does not expand the macro call within macro "y."

```
1:    make$change:
2:    DO;
3:    #g
4:    DECLARE coin(8) BYTE;
5:    DECLARE change BYTE;
6:    DECLARE I BYTE;
7:    next$coin:
8:    PROCEDURE(X);
9:    DECLARE X BYTE;
10:   coin(I) = X;
```

You can now delete line 3 from the file if you wish.

The exclamation point also allows you to examine each macro individually. For instance, suppose Macro "d" contains the following text.

```
DECLARE X = Y;
DECLARE Z = Y;
DECLARE N = M;
#b
```

and the Macro "b" which contains the following lines.

```
next$coin;
PROCEDURE(X);
DECLARE X BYTE;
#e
```

Macro "b" also contains a macro (e) that consists of the following lines.

```
making$change:
DO;
```

Suppose you want to use the first set of lines as a macro by itself. Rather than retyping the lines, you can append it to ED.MAC as follows (assuming you are editing ED.MAC).

```
*A
!d
.
*
```

Now when you invoke EDIT, ED.MAC expands the macro "d" without expanding the macros within "d."

Interpreting Commands In Macros

As a general rule, EDIT uses the backslash to remove special character characteristics from the character immediately following the backslash. An example is:

```
*S/end\./end,/
```

where "." tells edit you want to find the character string "end." rather than "end" followed immediately by any other character. Suppose your file contains the following lines.

```
ED SCHMIDT, 77 NW DRIVE, NEWBURG
JOHN SCHMIDT, 85 SW DRIVE, ORLANDO
SALLY SCHMIDT, 99 NE DRIVE, SPOKANE
SAMUEL SCHMIDT, 34 SE DRIVE, WOODVILLE
```

If the cursor is at line one, and you want to replace all the commas with semicolons, you can type:

```
*G;,;S;;\;;G
```

Your file now looks like this.

```
ED SCHMIDT; 77 NW DRIVE; NEWBURG
JOHN SCHMIDT; A85 SW DRIVE; ORLANDO
SALLY SCHMIDT; 99 NE DRIVE; SPOKANE
SAMUEL SCHMIDT; 34 SE DRIVE; WOODVILLE
```

Remember, when you use the Global command with a command list, EDIT temporarily saves the command list exactly as you typed it. In this case, EDIT saves the following command.

```
S;;\;;
```

Now, if you wish to place the previous Global Substitute command *in a macro,* you must add another backslash in front of the first backslash. This is because when EDIT stores the Global Substitute command in a macro buffer, it takes away a backslash.

Example

If you want to place the command "G;,;S;;\;;" in a macro, you must add another backslash to the command. The command in the macro internal buffer should look like the following illustration.

Command entered to macro:

```
*Um/S;,;s;;\\;;/
```

Command in macro buffer ***G;;;S;;\;;**

NOTE

If you get confused about the number of backslashes necessary in a macro, just examine the macro as follows.

U letter

Special Interpretations

The special character, carriage return (c/r), presents a unique exception to the general rule concerning macros and backslashes. In order to discuss the exception, we must go back to the previous example and use carriage returns.

In general the Global command interprets the command list portion exactly as you typed it, but the carriage return complicates the process. The carriage return normally terminates the command list portion of the Global command. Therefore, Global must know if you intend to place a carriage return in the command list buffer "\(c/r/)" or if you are using a carriage return to terminate the command list. This special interpretation of carriage return requires an extra backslash.

In other words, the Global command cannot pass a backslash and a carriage return to the buffer. This means a Global command with a command list intending to use "\(c/r)" must be written as "\\\(c/r)" so the Global command will interpret the command list as "\(c/r)".

Example

Suppose the following lines make up your file.

```
ED SCHMIDT, 77 NW DRIVE, NEWBURG
JOHN SCHMIDT, 85 SW DRIVE, ORLANDO
SALLY SCHMIDT, 99 NE DRIVE, SPOKANE
SAMUEL SCHMIDT, 34 SE DRIVE, WOODVILLE
```

If you want to replace all commas with carriage returns, you should type the following command.

```
*G:,:S::\\\(c/r)
:
```

EDIT actually saves "S::(c/r):" in temporary storage. This is because both the carriage return and its backslash require an extra backslash. The carriage return is just a special case that must have an extra backslash are executing a global command.

If you want to put the previous command in a macro, the process is more complex. You must add a backslash in front of each existing backslash and in front of the carriage return in order for the macro to function properly. You should type

```
*Ux/G:,:S::\\\\\\\(c/r)
:/
```

This command stores "G:,:S::\\\(c/r/)" in the macro buffer.

NOTE

If you get confused about the number of backslashes necessary in a macro, just examine the macro as follows.

U letter

APPENDIX A
ASCII CODES

Table A-1. ASCII Code List

Decimal	Octal	Hexa-decimal	Character
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<

Table A-1. ASCII Code List (Continued)

Decimal	Octal	Hexa-decimal	Character
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

APPENDIX B
IMPLEMENTATION PROBLEMS

Some systems do not respond to the tab key or the Tab option as depicted in previous chapters. Instead, you will *not* see the tabs as you type. Fear not! The tabs are included in the buffer but they do not become visible until you display the buffer.

INDEX

The primary reference of each multiple-page topic is in **boldface** type.

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

LS-004/3-5K/0782/WCP

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 NE Elam Young Parkway
Hillsboro OR 97123

ISO-N TECHNICAL PUBLICATIONS