

CHAPTER 12

OTHER BASIC STATEMENTS

This chapter discusses the READ, DATA, and RESTORE statements as well as some special MINC system functions.

Both assignment and INPUT statements are means of getting a value into a variable. Another method is via READ and DATA statements, which are explained in this section.

READ AND DATA STATEMENTS

INPUT statements enter pertinent but changing data into a program. READ and DATA statements are an alternate method of assigning values to variables. DATA statements are like a sequential file stored in the workspace and READ statements input the information from the DATA statements. READ and DATA statements are best used for generally defining values that are not going to change within one run of a program, but might change between runs.

For example, Chapter 14 describes a large program that begins by offering the program's users a menu of tasks to perform. The number of items in the menu usually does not change. However, if the program chooses to add a new task to the menu of tasks, then, for the next run, the number changes; however, it remains constant for the entire run. The number of items is best put in READ and DATA statements.

READ and DATA statements are always used together — that is, if one is in a program, the other must be somewhere in the same program. The form of the READ statement is:

READ variable-list

signed. If MINC finds a string variable where it expects a numeric variable, it outputs the following error message.

?MINC-F-DATA value or value from file does not match variable at line XX

where line XX is the line in which the READ statement could not be completed. If MINC finds a numeric value where it is expecting a string value, it simply assumes that the numbers are in the string and stores the ASCII values of the numbers in the string variable.

Integer values must match integer variables and real values must match real variables. You can put a numeric constant with no decimal point in either real or integer variables. For example, the following READ statement works.

```
10 READ A, A%
20 DATA 2,3
30 PRINT A,A%
RUNNH
```

2

3

However, the following READ statement does not work because the corresponding DATA value for Q (a real variable) is an integer literal.

```
20 READ Q,Q%
21 DATA 2%,3%
RUNNH
```

?MINC-F-DATA value or value from file does not match variable at line 20

Conversely, you cannot match a real literal to an integer variable. READ statements do not allow mixed modes as assignment statements do.

MINC ignores items in a DATA statement in excess of those used by the READ statements.

It is desirable to use DATA and READ statements rather than INPUT or assignment statements in a few cases. The first case, as mentioned earlier, is when you want to assign variable names to values that will remain constant within one run of a program, as in the menu example. Pertinent values in a program may actually be constants for an entire run of a program, but may need to be changed from run to run. You cannot use INPUT here because the program's users do not necessarily

know the correct value (such as the number of menu items). You might find it more convenient to use READ and DATA statements rather than assignment statements because with READ and DATA statements you need to change only the DATA statements rather than retyping entire assignment statements.

A second need of the READ and DATA statements occurs when the particular computer configuration has only one terminal for input and output. If the terminal is drawing an important graph or printing an important report, the question mark prompt from the INPUT statement might not be appropriate in the middle of the output. In this case, READ and DATA statements are more desirable because they do not affect the screen display.

The RESTORE Statement

You can use the RESTORE statement in conjunction with the READ and DATA statements. The RESTORE statement simply returns the DATA pointer to the first DATA item in the beginning of the whole DATA list, just as a RESTORE # statement restores a sequential file. For example, the following line causes the next READ after statement 50 to start reading from the very first DATA statement in the program, regardless of where the last DATA item was found.

A complete example is:

```
10 READ A$, B$
20 PRINT A$, B$
30 RESTORE
40 READ C$, B$, A$
50 PRINT C$, B$, A$
60 DATA ITEM1
70 DATA ITEM2,ITEM3,ITEM4
RUNNH
```

ITEM 1	ITEM 2	
ITEM 1	ITEM 2	ITEM 3

MINC SYSTEM FUNCTIONS

The MINC system functions change some of the characteristics of MINC, unlike the numeric and string functions discussed previously, which manipulate numbers or strings. These system functions are listed below. For descriptions of these functions and their forms, see Book 3.

```
TTYSET
RCTRLC
CTRLC
ABORT
RCTRLO
SYS
```

where the items in the variable list can be string or numeric variables separated by commas. The READ statement reads the values in the DATA statements (described below), and assigns these values to the variables in the variable list.

The form of the DATA statement is:

DATA data-list

where data-list contains the numbers or strings that you want to assign to the variables listed in the READ statement. Separate individual data items by commas. You do not have to enclose strings in quotation marks. For example:

```
10 READ A,B,A$,C
20 DATA 3,4.7,HELLO,9
30 PRINT A,B,A$,C
RUNNH
```

```
3                4.7                HELLO                9
```

When you type a RUN command, MINC searches for the first DATA statement and saves a pointer to its location. Each time MINC encounters a READ statement in the program, the next value in the DATA statement is assigned to the next variable in the READ list. If there are no more values in that DATA statement, MINC looks for the next DATA statement. If there are not enough DATA items, MINC prints the following error message and lists the line number of the READ statement that could not be finished.

```
?MINC-F-Too few values for INPUT or READ variables at line XX
```

The location of DATA statements within a program is arbitrary as long as the data items appear in the proper order. (You determine the proper order by what your program does.) It is a good practice to place all the DATA statements in a program together for quick reference when checking the program.

A READ statement assigns the next available element in a DATA statement to the first variable in its list. Then it assigns the next available element in a DATA statement to the next variable in its list until all variables have been satisfied. Again, the DATA statements are like a sequential file that is stored in the workspace.

The items in the DATA list must match the type of variable (string or numeric, integer or real) to which they will be as-

CHAPTER 13

FORMATTED OUTPUT

When the format as well as the content of your output is important, you can use the PRINT USING statement rather than the PRINT statement. The PRINT USING statement permits you to control the appearance and location of information on the output line, and thus enables you to create formatted lists, tables, reports, and forms.

With the PRINT USING statement, you set up a template that specifies the following kinds of numeric and string formats.

Numeric Output Format

- Number of digits
- Location of decimal point
- Special symbols:
 - trailing minus
 - asterisk fill
 - dollar sign
 - commas
- E notation

String Output Format

- Number of characters
- Left justification
- Right justification
- Centered
- Extended field

The general form of the PRINT USING statement is:

PRINT # channel, USING format-description, list

where:

format-description is a coded format template of the line to be printed. The format description is a string expression. If the format description is a string literal, it must be enclosed in quotation marks, not apostrophes.

list contains the items to be printed.

The format description sets up *fields* in the output line in which the items are printed. For example, the following two programs print a series of numbers and strings. One uses PRINT statements and the other uses PRINT USING statements.

PRINT

```
NEW PRINT
10 PRINT 1,'john smith'
20 PRINT 100,'jane doe'
30 PRINT 1.00000E+06,'jim brown'
40 PRINT 100.3,'lucy wong'
50 PRINT .0123456,'dave miller'
RUN
```

PRINT	27-MAR-80	11:32:11
1		john smith
100		jane doe
1.00000E+06		jim brown
100.3		lucy wong
.0123456		dave miller

READY

PRINT USING

```
NEW USING
10 A$="#####.## 'RRRRRRRRRRRR'"
20 PRINT USING A$,1,'john smith'
30 PRINT USING A$,100,'jane doe'
40 PRINT USING A$,1.00000E+06,'jim brown'
50 PRINT USING A$,100.3,'lucy wong'
60 PRINT USING A$,,.0123456,'dave miller'
```

RUN

USING	27-MAR-80	11:32:57
-------	-----------	----------

1.00	john miller
100.00	jane doe
1,000,000.00	jim brown
100.30	lucy wong
0.01	dave miller

READY

Notice that the numbers printed with the PRINT USING statement are aligned by the decimal point and the names are right justified in their field.

The format description for this example is:

"#####.## 'RRRRRRRRRRRR"

NOTE

The format description describes the entire line of output. When you use a PRINT USING statement, the print zones are no longer in effect, and commas and semi-colons do nothing but separate list items.

You specify the *number of digits* in the format description with a number sign (#). Thus, a format description for a 3-place number is "###". If the number is negative, you must have a # for the minus sign too. Thus, "-###" is the format description for a 3-place positive number or a 2-place negative number. If the format does not have enough places for all of the digits in the number, MINC prints out a percent sign (%) and then prints the number with a PRINT statement format. For example:

```
PRINT USING "####", 1234,-123,-1234
1234
-123
%-1234
```

You specify the location of a *decimal point* in the numeric field by placing a period (.) in the appropriate place in the format description. For example, the format description for a number with 5 places to the left of the decimal point and 2 places to the right is "#####.##". If there are not enough places in the format description after the decimal point, MINC rounds the number. For example:

```
PRINT USING ".###",.999
.999
```

FORMATTING NUMERIC OUTPUT

mas, 4 places before the decimal point, 2 places after and a trailing minus sign. You cannot combine a dollar sign with asterisk fill or a preceding minus. For example:

```
PRINT USING "$$#,##.##-", 1234.56, -1234.56
$1,234.56
$1,234.56-
```

To print a number using *E notation*, you place 4 carets (^^^) after the number signs. The 4 carets reserve space for the E, followed by a plus or minus sign, and the 2-digit exponent. In E notation, the digits to the left of the decimal point are not filled with spaces. Instead, the first nonzero digit is shifted to the leftmost place, and the exponent is equal to the number of places that the decimal point is shifted from the number in standard notation. For example:

```
PRINT USING "##.####^^^", 1234.56, -1234.56
12.3456E+02
-1.2346E+03
```

The following example shows all the forms of numeric format. Notice that in line 100, the program tries to print a negative number in asterisk fill without a trailing minus sign.

```
10 P = 1234.56
20 N = -1234.56
30 PRINT USING "#####.##", P, N
35 PRINT
40 PRINT USING "####.##", P, N
45 PRINT
50 PRINT USING "***#####.##", P
55 PRINT
60 PRINT USING "**,#####.##", P
65 PRINT
70 PRINT USING "**,#####.##-", P, N
75 PRINT
80 PRINT USING "$$#,##.##-", P, N
85 PRINT
90 PRINT USING "##.####^^^", P, N
95 PRINT
100 PRINT USING "***#####.##", N
RUN
```

```
NONAME          27-MAR-80          15:03:24
```

```
1234.56
-1234.56
```

```
1234.56
%-1234.56
```

```

      A
     AB
    ABC
   ABCDE

```

READY

To print strings in a *centered* field, use a C in the format description for each character in the field after the first. For example:

```
PRINT USING "CCCCC", 'A', 'AB', 'ABC', 'ABCDE'
```

```

      A
     AB
    ABC
   ABCDE

```

READY

Notice that if the string cannot be exactly centered (such as a two-character string in a seven-character field), MINC prints the string one character off center to the left.

To print strings in an *extended* field, use an E in the format description for each character in the field after the first. The extended field is the only field that ensures the printing of the entire string. If you specify an extended field, MINC left-justifies the string as it does for a left-justified field. But, if the string has more characters than there are places in the field, MINC extends the field and prints the entire string. For example:

LIST

```
NONAME          25-JUL-80          11:36:44
```

```

10 DIM A$(5)
20 PRINT 'input 5 strings of varying length'
30 FOR I=1 TO 5 \ INPUT A$(I) \ NEXT I
40 PRINT \ PRINT 'centered  extended  left    right'
50 PRINT '-----'
70 F$="..CCCC.  ..EEEE.  ..LLLL.  ..RRRR."
80 FOR I=1 TO 5
90 PRINT USING F$,A$(I),A$(I),A$(I),A$(I),A$(I)
100 NEXT I
110 END

```

READY
RUN

```
NONAME          25-JUL-80          11:36:49
```

```
PRINT USING ".##",.994
.99
```

```
PRINT USING ".##",.999
%.999
```

```
PRINT USING "#.##",.999
1.00
```

If you would like *commas* to appear in numeric output, place a comma anywhere in the numeric format description before the decimal point. Then, commas will print every three places. For example, the format description for a number with 7 places to the left of the decimal point, 2 places to the right, and commas is "`##,####.##`". Note that the comma can appear anywhere before the decimal point. For example:

```
PRINT USING "##,##.##", 1234.56
1,234.56
```

If you would like the *minus sign* after the number rather than before, place a minus sign after the number signs in the format description. For example, if you want a numeric field with 3 places to the left of the decimal point, 2 places to the right, and a trailing minus sign, use "`###.## -`" as the format description. If the number is negative, the minus sign will follow it. If the number is positive, there will be no minus sign. For example:

```
PRINT USING "#,###.##-", 1234.56, -1234.56
1,234.56
1,234.56-
```

If you would like the numeric field *filled with preceding asterisks* (*), place 2 asterisks as the first 2 places of the format description. The asterisks also define two places in the field. For example, "`***##.##-`" defines a numeric field with asterisk fill, 4 places before the decimal point, 2 places after, and a trailing minus. You cannot combine asterisk fill with a preceding minus. For example:

```
PRINT USING "***###.##-", 1234.56, -1234.56
**1234.56
**1234.56-
```

If you would like the numeric field to have a dollar sign (\$), place 2 dollar signs as the first 2 places of the format description for that field. The dollar signs define one place in the field for a dollar sign and one place for the first number. For example, "`$$,###.##-`" defines a numeric field with a dollar sign, com-

```
***1234.56
**1,234.56
```

```
**1,234.56
**1,234.56-
```

```
$1,234.56
$1,234.56-
```

```
12.3456E+02
-1.2346E+03
```

?MINC-F-Invalid PRINT USING format or syntax at line 100

READY

The format description in statement 100 caused an error message because the asterisk fill format description left no room for a trailing minus sign.

FORMATTING STRING OUTPUT

A string format description starts with an apostrophe ('). If you want to print a string field with one character, the format description is "'".

If a string is larger than its specified string field, MINC prints as much of the string as fits in the field and ignores the rest. The only exception is that for extended fields (described below), MINC prints the entire string.

To print strings in a *left-justified* field, use an L in the format description for each character in the field after the first. (Remember that the first character is denoted by an apostrophe.) For example:

```
10 PRINT USING "'LLL", 'ABCDE'
20 PRINT USING "'LLLLLL", 'ABCDE'
RUNNH
```

```
ABCD
ABCDE
```

READY

To print strings in a *right-justified* field, use an R in the format description for each character in the field after the first. For example:

```
PRINT USING "'RRRRRR", 'A','AB','ABC','ABCDE'
```

input 5 strings of varying length

? a
? ab
? abc
? abcd
? abcdefghij

centered	extended	left	right
.. a .	.. a .	.. a .	.. a .
.. a b .	.. a b .	.. a b .	.. a b .
.. a b c .	.. a b c .	.. a b c .	.. a b c .
.. a b c d .	.. a b c d .	.. a b c d .	.. a b c d .
.. a b c d e .	.. a b c d e f g h i j .	.. a b c d e .	.. a b c d e .

READY

The underlined field has been extended. Note that the rest of the line is displaced five places.

PRINT USING STATEMENT ERROR CONDITIONS

There are two types of PRINT USING error conditions: fatal and nonfatal.

When a fatal error occurs, MINC stops executing the program and prints the following message.

?MINC-F-Invalid PRINT USING format or syntax

When a nonfatal error occurs, MINC continues to execute the program, although the resulting output may not be in the format intended.

See Book 3 for the details of PRINT USING error conditions.

CHAPTER 14

COMBINING PROGRAMS

When you are writing a program that solves a complex problem, that program can become very long. You might want to break the program into segments for two reasons: the program is too long and complicated to think about as a whole, or the program and its arrays are too long to fit in the workspace.

There are two statements and one command that help you to break a program into segments.

- | | |
|---------|--|
| CHAIN | When one segment is finished executing, the CHAIN statement at the end of the segment brings the next segment into the workspace. |
| APPEND | The APPEND command merges program segments together in the workspace. |
| OVERLAY | In the immediate mode, the OVERLAY statement works like the APPEND command. It also works in the program mode, merging two program segments together in the workspace. |

This chapter describes these statements in more detail. For further explanation, see also Book 3.

CHAINING PROGRAMS TOGETHER

You can use the CHAIN statement to break a program into segments. You create each segment as you would create any program, except that you end each segment (except the last to be executed) with a CHAIN statement.

The form of the CHAIN statement is:

CHAIN 'filespec' LINE number

where:

'filespec' is a string representation of the next segment. The filespec is of the form:

dev:name.typ

If you leave off the device, MINC assumes SY0:. If you leave off the file type, MINC first tries to find a file with type .BAC, then it looks for a file with type .BAS.

number is optional (as well as LINE). If LINE and number are present, they represent the statement number at which MINC starts execution of the next segment. If LINE and number are not present, MINC begins execution of the segment at the lowest numbered statement. For more information about the LINE number argument of the CHAIN statement, see Book 3.

The CHAIN statement works faster with compiled segments.

When MINC executes the CHAIN statement, it does the following:

1. Closes any open files (virtual array or sequential).
2. Changes the workspace name to the name in the CHAIN file specification.
3. Deletes the current program segment from the workspace, including:

all program lines

all variables not defined in COMMON (explained in a later section)

all arrays not defined in COMMON

all user-defined functions

4. Loads the new segment into the workspace and executes it.

Use of the CHAIN statement is shown by an example in the following section. Further explanation of the CHAIN statement — that is, saving variables in COMMON — is given after the example.

NOTE

The following program segments are stored on the demonstration diskette in files FILEMT.BAS, FILEM1.BAS, FILEM2.BAS, FILEM3.BAS, FILEM4.BAS, and FILEM5.BAS. If you do not have a demonstration diskette, copy the Master diskette using the instructions given in Part II of Book 1. Running these programs might require use of the EXTRA_SPACE command. See the EXTRA_SPACE command in Book 3.

Example — A Sequential File Maintenance Program

Suppose you want to make managing sequential files easier for yourself and your colleagues. You can write a program to do the types of file manipulation procedures that are quite common, such as

- inputting a sequential file
- sorting the contents of a sequential file
- merging two ordered files into one ordered file
- concatenating two files
- listing a file on the screen

Writing a program to do all of these things is quite complicated. So, you can write a series of small programs, one for each item in the list, and chain them together.

Once you know what you want to do, you can write a “main program” that manages your other programs. For example, the following program manages the sequential file maintenance program. It gives its users a choice of what to do with their files, and when they choose what to do, the program chains to the right choice.

Lines 10 through 100 print the choices on the screen. Line 160 inputs the user's choice. Line 170 checks to see that the choice is within the range, and then chains to the program that matches the choice.

This program's name is FILEMT. The program representing choice 1 is FILEM1, the program representing choice 2 is FILEM2 and so forth.

LIST

```
FILEMT                16-MAY-80                14:21:33

10 FOR I=1 TO 23 \ PRINT \ NEXT I \ REM - Clear Screen
20 PRINT 'SEQUENTIAL FILE MAINTENANCE PROGRAM'
30 PRINT
40 PRINT 'Enter program number from:' \ PRINT
50 PRINT '1          Input a file (NOTE: use editor to update)'
60 PRINT '2          Sort a file'
70 PRINT '3          Merge 2 files'
80 PRINT '4          Concatenate files'
90 PRINT '5          List a file on the screen'
100 PRINT '6         Exit file maintenance program'
110 PRINT
120 REM - S is the second to the last choice
130 REM - L is the last choice
140 READ S,L
150 DATA 5,6
160 INPUT I
170 IF I <= I THEN IF I <= S THEN CHAIN 'filem' + STR$(I)
180 IF I=L GO TO 210
190 PRINT 'Enter number between 1 and ';L;
200 GO TO 160
210 PRINT 'Exit File Maintenance Program'
220 END
```

READY

The variables S and L are used to represent the second to last choice and the last choice that appear on the screen. By using the variables in a READ statement, if you want to add a choice to the list, you have to change only the DATA statement and add the appropriate print statements.

By creating the FILEMT program, you can now write one program for each task. The problem is not so difficult when it is broken into segments.

The program FILEM1, representing choice 1, is shown below. Notice that FILEM1 chains back to the main program, FILEMT, to offer the user the next choice.

LIST

FILEM1 16-MAY-80 13:21:53

```

10 FOR I=1 TO 23 \ PRINT \ NEXT I \ REM -- Clear screen
20 PRINT 'FILE INPUT PROGRAM' \ PRINT \ PRINT
30 PRINT 'Input file name'; \ LINPUT F$
40 PRINT 'At each question mark, type the next line of the input file.'
50 PRINT 'To end the file, press the RETURN key at the question mark.'
60 OPEN F$ FOR OUTPUT AS FILE #1
70 LINPUT N$
80 IF N$="" GO TO 110
90 PRINT #1,N$
100 GO TO 70
110 CLOSE #1
120 PRINT 'File input complete'
130 CHAIN 'filemt'
140 END

```

READY

The program for choice 2, FILEM2, is shown below. This program sorts a sequential file. The program restricts the file to be a maximum of 100 records with a maximum of 10 lines per record. For example, a file with 50 names and addresses (3 lines each) has 50 records with 3 lines per record. That is, each name and address represents 1 record.

Lines 1 through 70 input the pertinent information about the file to be sorted. Lines 80 through 150 enter the file into the workspace. Lines 4000 through 4150 sort the file now stored in array N. For the explanation of the sorting algorithm, the shell sort, see *The Art of Computer Programming, Volume 3/Sorting and Searching* by Donald E. Knuth, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973. Finally, Lines 4170 through 5060 store the sorted file. Then the program chains back to the main program, FILEMT.

LIST

FILEM2 16-MAY-80 16:52:48

```

1 REM - N$(N,I) is the array to be sorted. N is the number of elements
3 REM - that are i lines long.
5 DIM N$(100,10)
10 FOR I=1 TO 23 \ PRINT \ NEXT I
20 PRINT 'FILE SORT PROGRAM'
30 PRINT \ PRINT
40 PRINT 'Name of file to be sorted'; \ INPUT F$
50 OPEN F$ FOR INPUT AS FILE 1
60 PRINT 'How many lines represent one record'; \ INPUT L

```

PROGRAMMING FUNDAMENTALS

```
65 L=L-1
70 N=1
74 REM
75 REM -----
76 REM - Input the file into the workspace
77 REM
80 IF END #1 GO TO 140
90 FOR I=0 TO L
100 LINPUT #1,N$(N,I)
110 NEXT I
120 N=N+1
130 GO TO 80
140 N=N-1
150 CLOSE 1
3000 REM -----
3010 REM - Sort the file
3020 REM
3030 REM - For an explanation of the shell sort see:
3040 REM - Knuth, D.E., "The Art of Computer Programming,
3050 REM - Volume 3 / Sorting and Searching", Addison-Wesley Publishing
3060 REM - Company, Reading, Massachusetts, 1973
3065 REM
3070 REM -----
4000 REM - shell sort
4005 REM
4010 G=N
4020 G=INT(G/2)
4030 K=G
4040 K=K+1
4050 FOR I=0 TO L \ V$(I)=(K,I) \ NEXT I
4060 J=K
4070 J1=J
4080 J=J-G
4090 IF J<1 GO TO 4130
4100 FOR I=0 TO L
4101 IF V$(I)<N$(J,I) GO TO 4110
4102 IF V$(I)>N$(J,I) GO TO 4130
4103 NEXT I
4110 FOR I=0 TO L \ N$(J1,I)=N$(J,I) \ NEXT I
4120 GO TO 4070
4130 FOR I=0 TO L \ N$(J1,I)=V$(I) \ NEXT I
4140 IF K<N GO TO 4040
4150 IF G>1 GO TO 4020
4155 REM
4160 -----
4165 REM
4170 PRINT 'Name of output file to hold sorted file'; \ INPUT F$
5000 OPEN F$ FOR OUTPUT AS FILE 1
5010 FOR J=1 TO N
5020 FOR I=0 TO L
5030 PRINT #1,N$(J,I)
5040 NEXT I
5050 NEXT J
5060 CLOSE 1
5070 PRINT 'File sort complete'
```

```
5080 CHAIN 'filemt'
5090 END
```

READY

The program FILEM3, merges two sorted files into one larger sorted file, removing duplicates. This program is a generalized version of the mailing label program described in Chapter 11, pages 154–156.

Lines 10 through 100 input the pertinent information about the file. The rest of the program merges the two files. For further explanation of the algorithm, refer back to Chapter 11. Finally, the program chains back to FILEMT for the user's next choice.

LIST

```
FILEM3          16-MAY-80          16:29:23
```

```
10 FOR I=1 TO 23 \ PRINT \ NEXT I
20 PRINT 'FILE MERGING PROGRAM'
30 PRINT \ PRINT
40 PRINT 'Input first file name'; \ INPUT F1$
50 PRINT 'Input second file name'; \ INPUT F2$
60 PRINT 'How many lines represent one record'; \ INPUT L
70 PRINT 'output file name'; \ INPUT O$
80 OPEN F1$ FOR INPUT AS FILE 1
90 OPEN F2$ FOR INPUT AS FILE 2
100 OPEN O$ FOR OUTPUT AS FILE 3
104 REM
105 REM -----
106 REM - Process file 1
107 REM
110 IF END #1 GO TO 130
120 GO TO 160
130 F=2 \ REM - End of file 1; file 2 left
140 GO TO 540
150 REM - Get next record from file 1
160 FOR I=1 TO L \ LINPUT #1,I1$(I) \ NEXT I
164 REM
165 REM -----
166 REM - Process file 2
167 REM
170 IF END #2 GO TO 190
180 GO TO 240
190 F=1 \ REM - End of file 2; file 1 left
200 REM - Get record left over from file 1
210 FOR I=1 TO L \ I$(I)=I1$(I) \ NEXT I
220 GO TO 560
230 REM - Get next record from file 2
240 FOR I=1 TO L \ LINPUT #2,I2$(I) \ NEXT I
250 GO TO 340
255 REM
```

PROGRAMMING FUNDAMENTALS

```
256 REM -----
257 REM - Process file 1
258 REM
260 IF END #1 GO TO 280
270 GO TO 330
280 F = 2 \ REM - End of file 1: file 2 left
290 REM - Get record left over from file 2
300 FOR I = 1 TO L \ I$(I) = I2$(I) \ NEXT I
310 GO TO 560
320 REM - Get next record from file 1
330 FOR I = 1 TO L \ INPUT #1, I1$(I) \ NEXT I
334 REM
335 REM -----
337 REM - Decide which record is next in the output file
338 REM
340 FOR I = 1 TO L
350 IF I1$(I) >= I2$(I) GO TO 400
355 REM
360 REM - Process the record from file 1
365 REM
370 FOR J = 1 TO L \ PRINT #3, I1$(J) \ NEXT J
380 GO TO 260
400 IF I2$(I) >= I1$(I) GO TO 450
405 REM
410 REM - Process the record from file 2
415 REM
420 FOR J = 1 TO L \ PRINT #3, I2$(J) \ NEXT J
430 GO TO 170
450 NEXT I
460 REM - They are equal records — so print from file 1
470 FOR J = 1 TO L \ PRINT #3, I1$(J) \ NEXT J
480 GO TO 110
525 REM
526 REM -----
527 REM - finish off remaining file
530 REM
540 IF END #F GO TO 580
550 FOR I = 1 TO L \ INPUT #F, I$(I) \ NEXT I
560 FOR I = 1 TO L \ PRINT #3, I$(I) \ NEXT I
570 GO TO 540
580 CLOSE 1,2,3
590 PRINT 'File merge complete'
600 CHAIN 'filemt'
610 END
```

READY

FILEM4 concatenates files; that is, it creates a large output file that holds all of the input files, one after another. Finally FILEM4 chains back to FILEMT.

LIST

```

10 FOR I=1 TO 23 \ PRINT \ NEXT I
20 PRINT 'FILE CONCATENATION PROGRAM'
30 PRINT \ PRINT
35 PRINT 'Output file name'; \ INPUT O$
37 OPEN O$ FOR OUTPUT AS FILE #12
60 PRINT 'Next file name'; \ INPUT F$
70 OPEN F$ FOR INPUT AS FILE 1
80 IF END #1 GO TO 1000
90 LINPUT #1,L$
100 PRINT #12,L$
110 GO TO 80
1000 CLOSE 1
1010 PRINT 'Merge another file (Y or N)'; \ INPUT R$
1015 IF R$='Y' GO TO 60
1016 IF R$='y' GO TO 60
1020 CLOSE 12
1030 PRINT 'File merge complete'
1040 CHAIN 'filemt'
1050 END

```

READY

Lastly, FILEM5 lists a file on the screen.

LIST

```

FILEM5          16-MAY-80          14:30:52

```

```

10 FOR I=1 TO 23 \ PRINT \ NEXT I \ REM - Clear screen
20 PRINT 'FILE LIST PROGRAM ' \ PRINT \ PRINT
30 PRINT 'File name (default type is .DAT)'; \ INPUT F$
40 OPEN F$ FOR INPUT AS FILE 1
50 IF END #1 GO TO 90
60 LINPUT #1,L$
70 PRINT L$
80 GO TO 50
90 CLOSE 1
100 PRINT 'List another file (Y or N)'; \ INPUT R$
110 IF R$="Y" GO TO 30
120 IF R$='y' GO TO 30
130 CHAIN 'filemt'
140 END

```

READY

To run this program, type:

RUN FILEMT

Notice how much simpler programming the file maintenance program became when it was segmented. Most programs can be designed this way. Later, if you find that you want to recombine

the segments into one large program again, you can do so easily, using the APPEND command (described later in this chapter).

Preserving Values of Variables in a Chain

The COMMON statement preserves the values of variables and arrays when one BASIC program segment chains to another. Any variables or arrays listed in COMMON statements retain the same variable names and values after the CHAIN is executed.

The form of the COMMON statement is:

COMMON list

where list is the list of variables and arrays separated by commas. For example:

```
10 COMMON A$, F%, C2, V(100), I$(10,3)
```

You must specify the dimensions of arrays in COMMON. The COMMON statement replaces the DIM statement for arrays stored in COMMON.

When MINC brings in the new program segment, it checks to see that the new segment has corresponding COMMON statements. The lists in the COMMON statements of the new segments must contain the same variable names, data types, and array dimensions in the same order as the lists in the previous segment. You can change the line numbers and the number of items specified in each COMMON statement, but you cannot change the order of the variables and arrays.

For example:

<i>Segment 1</i>	<i>Segment 2</i>
10 COMMON A,B,C\$	10 COMMON A,B
20 COMMON D(100)	30 COMMON C\$,D(100),G\$(2)
30 COMMON G\$(2)	
	<i>Segment 3</i>
	10 COMMON A,B,D(100)
	20 COMMON C\$
	30 COMMON G\$(2)

Program segments 1 and 2 contain equivalent COMMON statements. Segment 3 however does not contain equivalent COMMON statements because D(100) appears before C\$.

If in the new segment you do not list the variables and arrays in COMMON statements as in the original, MINC prints the following error message and stops program execution.

?MINC-F-COMMON variables not in the same order as in last program at line 10

Below is a short example to demonstrate the COMMON statement.

The file named SEG1 is listed below. Line 10 preserves array I(100) in COMMON. The program segment assigns the values 2 through 200 to both arrays I and J and then chains to SEG2.

```
10 COMMON I(100)
20 DIM J(100)
30 PRINT 'Executing SEG1'
40 FOR K = 1 TO 100
50 I(K) = K*2
60 J(K) = K*2
70 NEXT K
80 CHAIN 'SEG2'
90 END
```

The file named SEG2 is listed below. Line 10 preserves I(100) in COMMON. This program segment sums all of the elements of each array; that is, T1 holds the sum of the elements of array I and T2 holds the sum of the elements of array J.

```
10 COMMON I(100)
20 DIM J(100)
30 PRINT 'Executing SEG2'
40 FOR K = 1 TO 100
50 T1 = T1 + I(K)
60 T2 = T2 + J(K)
70 NEXT K
80 PRINT 'The sum of array I is';T1
90 PRINT 'The sum of array J is';T2
100 END
```

A run of this program produces the following output.

```
run seg1
Executing SEG1
Executing SEG2
The sum of array I is 10100
The sum of array J is 0
```

READY

Note that MINC preserves I(100) but does not preserve J(100) since array J was not saved in COMMON.

APPENDING PROGRAMS

For more detail of the COMMON statement, see Book 3.

The APPEND command merges the specified file with the program already in the workspace. The resulting program in the workspace is a combination of the two programs. This command is especially useful when you wish to add subroutines from a diskette to a new program in the workspace.

When you use the APPEND command, you must be careful that the statement numbers are aligned properly. For example, suppose this simple program is in the workspace.

```
10 PRINT 'original program line 10'  
30 PRINT 'original program line 30'  
40 PRINT 'original program line 40'
```

Below is a program stored in APND.BAS.

```
20 PRINT 'APND line 20'  
40 PRINT 'APND line 40'  
50 END
```

Now, if you type:

```
APPEND APND
```

The following program ends up in the workspace.

```
10 PRINT 'original program line 10'  
20 PRINT 'APND line 20'  
30 PRINT 'original program line 30'  
40 PRINT 'APND line 40'  
50 END
```

Line 40 of the original program was superseded by line 40 of APND.BAS.

Note that the APPEND command does not affect the workspace name.

The form of the APPEND command is:

```
APPEND filespec
```

where filespec is optional and is of the form:

```
dev:name.typ
```

If you leave out the device, MINC assumes SY0:. If you leave out

the file type, MINC assumes .BAS. You cannot APPEND a compiled program. If you leave out the filespec, MINC prints:

OLD FILE NAME—

to which you must enter the file specification.

You can use the APPEND command to make one program out of a group of segments. For example, the file maintenance program was easier to design and program by breaking the problem into segments. However, the CHAIN command works slowly because it must retrieve each segment from the diskette. Thus, now you might want to merge the file maintenance segments into one program that fits into the workspace all at once.

To do this, you must make some minor modifications to the segments. In general these modifications are as follows. First, make copies of all the segments before you alter them. Then change FILEMT.BAS so that it does not use the CHAIN command (because you are altering the program so that it does not chain). Use ON/GO TO instead of CHAIN. Then, resequence each segment so that they do not have conflicting statements. Finally, append all the segments together. The following sequence shows you specifically how to modify FILEMT and FILEM1. You can modify segments FILEM2 through FILEM5 similarly to FILEM1.

Below is another listing of the main program, FILEMT.

```

10 FOR I=1 TO 23 \ PRINT \ NEXT I \ REM - Clear Screen
20 PRINT 'SEQUENTIAL FILE MAINTENANCE PROGRAM'
30 PRINT
40 PRINT 'Enter program number from:' \ PRINT
50 PRINT '1      Input a file (NOTE: use editor to update)'
60 PRINT '2      Sort a file'
70 PRINT '3      Merge 2 files'
80 PRINT '4      Concatenate files'
90 PRINT '5      List a file on the screen'
100 PRINT '6      Exit file maintenance program'
110 PRINT
120 REM - S is the second to the last choice
130 REM - L is the last choice
135 RESTORE
140 READ S,L
150 DATA 5,6
160 INPUT I
170 IF 1 <= I THEN IF I <= S THEN CHAIN 'filem' + STR$(I)
180 IF I=L GO TO 210
190 PRINT 'Enter number between 1 and 'L;
200 GO TO 160

```

```
210 PRINT 'Exit File Maintenance Program'
220 END
```

First, save FILEMT in FILEMT.OLD. Now you have a copy of the original program before you alter it.

To make one large program out of all the segments, you must change lines 170 and 220 as follows.

```
170 IF 1<=I THEN IF I<=5 THEN ON I GO TO 1000,2000,3000,4000,5000

220 GO TO 32767
```

Rather than chaining to a segment, line 170 has been changed to go to the appropriate part of the program with an ON/GO TO statement. Line 32767 is now the END statement. Notice that a new line, 135, has been added. You must restore the DATA statement now, every time the READ statement is executed. Previously, the DATA statement was restored every time the FILEMT.BAS program was chained to. Now that the menu part of the program remains in the workspace at all times, you must specifically restore the DATA statement.

Now you can save this new version in the file called FILEMT.BAS. The old version is stored in FILEMT.OLD.

Now load FILEM1.BAS with the OLD command and save it in FILEM1.OLD before you change it.

Below is another listing of FILEM1.

```
10 FOR I=1 TO 23 \ PRINT \ NEXT I \ REM — Clear screen
20 PRINT 'FILE INPUT PROGRAM' \ PRINT \ PRINT
30 PRINT 'Input file name'; \ LINPUT F$
40 PRINT 'At each question mark, type the next line of the input file.'
50 PRINT 'To end the file, press the RETURN key at the question mark.'
60 OPEN F$ FOR OUTPUT AS FILE #1
70 LINPUT N$
80 IF N$="" GO TO 110
90 PRINT #1,N$
100 GO TO 70
110 CLOSE #1
120 PRINT 'File input complete'
130 CHAIN 'FILEMT'
140 END
```

First resequence FILEM1 with the following command:

```
RESEQ 1000
```

This command resequences FILEM1 so that it starts with line 1000. It must start with line 1000 because the new FILEMT program transfers control to 1000 if the choice is 1 (line 170 of the new FILEMT).

Now you must change lines 1120 and 1130. Line 1120 should be changed to

```
1120 GO TO 10
```

Now when the program is done with inputting a file, it goes back to the beginning and displays the choices again.

Line 1130 should be deleted. You cannot have an END statement in the middle of a program (line 32767 is now the END statement).

Now you can change the workspace name to FILEMT with the RENAME command and APPEND FILEMT. The resulting program is listed below.

LIST

```
FILEMT          18-MAY-80          02:36:27

 10 FOR I=1 TO 23 \ PRINT \ NEXT I
 20 PRINT 'SEQUENTIAL FILE MAINTENANCE PROGRAM'
 30 PRINT
 40 PRINT 'Enter program number from:' \ PRINT
 50 PRINT '1      Input a file (NOTE: use editor to update)'
 60 PRINT '2      Sort a file'
 70 PRINT '3      Merge 2 files'
 80 PRINT '4      Concatenate files'
 90 PRINT '5      List a file on the screen'
100 PRINT '6      Exit file maintenance program'
110 PRINT
120 REM - S is the second to the last choice
130 REM - L is the last choice
135 RESTORE
140 READ S,L
150 DATA 5,6
160 INPUT I
170 IF 1<=I THEN IF I <=S THEN ON I GO TO 1000,2000,3000,4000,5000
180 IF I=L GO TO 210
190 PRINT 'Enter number between 1 and 'L;
200 GO TO 160
210 PRINT 'Exit File Maintenance Program'
220 GO TO 32767
1000 FOR I=1 TO 23 \ PRINT \ NEXT I \ REM - Clear Screen
1010 PRINT 'FILE INPUT PROGRAM' \ PRINT \ PRINT
1020 PRINT 'Input file name'; \ INPUT F$
```

```
1030 PRINT 'At each question mark, type the next line of the input file.'
1040 PRINT 'To end the file, press the RETURN key at the question mark.'
1050 OPEN F$ FOR OUTPUT AS FILE #1
1060 LINPUT N$
1070 IF N$=" GO TO 1100
1080 PRINT #1,N$
1090 GO TO 1060
1100 CLOSE #1
1110 PRINT 'File input complete'
1120 GO TO 10
32767 END
```

READY

Now you must perform similar modifications to FILEM2 through FILEM5. Then your whole file maintenance program will be one large program and will work faster than the chained version.

OVERLAYING A PROGRAM

The OVERLAY statement works similarly to the APPEND command. However, the OVERLAY statement can be used within a program in the program mode. Thus, the OVERLAY statement can be very tricky to use if you are interweaving statements from the workspace and from the specified file.

You can use the OVERLAY statement to load segments of programs into the workspace, similarly to the way you use the CHAIN statement. The OVERLAY statement allows easier communication between segments than the CHAIN statements. When you use the OVERLAY statement, the values of all variables and arrays are preserved and all open files remain open. However, you must ensure that the line numbers of the segments merge correctly, which you need not do with CHAIN.

The form of the OVERLAY statement is:

OVERLAY 'filespec' LINE number

where:

'filespec' is a string representation of the file specification of the overlay segment. The filespec is of the form:

dev:name.typ

If you leave off the device, MINC defaults to SY0:. If you leave off the file type, MINC de-

faults to .BAS. (Note: you cannot overlay a compiled program.)

number is optional (as well as LINE). If LINE and number are present, they represent the statement number at which MINC starts execution after the overlay. If you omit LINE and number, MINC starts execution at the next sequential statement number after the OVERLAY statement.

Note that if you enter the OVERLAY statement on a multi-statement line, MINC ignores the rest of the line.

For more information about the OVERLAY command, see Book 3.

In MINC you perform this calculation by typing in the following BASIC command:

```
PRINT (27 + 32)*(15-8)/327
```

MINC displays the answer beneath the PRINT statement, like this:

```
PRINT (27 + 32)*(15-8)/327
1.263
```

When you use MINC as a calculator, you are using it in what is called the *immediate mode*. In the immediate mode, MINC performs your instructions as soon as you press RETURN.

The rules for using MINC as a calculator are quite simple and are explained in this chapter.

THE PRINT STATEMENT

To get MINC to display information on the terminal, you must use the PRINT statement. As you saw above, the form of the PRINT statement is:

```
PRINT expression (RET)
```

where expression represents the number or calculation that is to be printed, and (RET) represents pressing the RETURN key. The expression is printed in blue ink, because it is optional in a PRINT statement. If you omit the expression, MINC prints a blank line.

The following three examples are valid PRINT statements where 7, 23.8, and the result of $5324.7 + 78625.9$ are to be printed by each PRINT statement respectively.

```
PRINT 7
```

```
PRINT 23.8
```

```
PRINT 5324.7 + 78625.9
```

If you enter each of the above statements, MINC prints the results on the screen as follows:

```
PRINT 7
7
```

CHAPTER 15

KEYPAD EDITING WITH MINC

In the immediate and program modes, MINC interprets each line you type as a statement or command in the BASIC language. When you are using or writing programs, the BASIC commands and statements are adequate.

However, neither the program mode nor the immediate mode allows you to easily type, correct, store, or display ASCII files that are not program files. For example, Chapters 11 and 13 demonstrated a simple program that allows you to type in an ASCII file such as names and addresses (INFILE and FILEM1). If you make a typing mistake in one of these files, however, you can correct it only by writing a program in the program mode.

MINC's alternate mode of operation is editing. By working with the keypad editor, you can create, inspect, or modify any ASCII file. You can use the editor to type in BASIC programs, but in doing so, you can create programs that will produce serious errors in BASIC. The keypad editor is far more suitable for the following sorts of files:

- Sequential data files (for INPUT and LINPUT statements).
- Memos and letters.
- Charts.
- Documentation for special programs and equipment you use.
- Any other files that have only ASCII characters.

You cannot use the editor to create, inspect, or modify four kinds of files:

- Files with the protected file types .SYS, .SAV, .COM or .BAD.
- Compiled program files — .BAC is the default file type MINC uses for these.
- Virtual array files — .DAT is the default file type MINC uses.
- Any other files that have non-ASCII characters.

The keypad editor is a useful tool for MINC users who already have experience with MINC's immediate and program modes or with text editing programs on other equipment. As you will soon see as you learn how the editor works, its chief advantage is that it continuously displays the file that you are working with. You can scan downward and upward freely; you always see a 24-line part of your file with the line you are changing in the middle of the screen. You can also erase characters, insert characters at any position on a line, and search forward or backward for character strings that are elsewhere in the file.

THE THREE EDITING COMMANDS

The three MINC commands that run the keypad editor are:

- | | |
|---------|---|
| INSPECT | Use INSPECT when you want to look at an existing ASCII file but you do not want to change the file in any way; INS is the valid abbreviation. |
| EDIT | Use EDIT when you want to add to an existing ASCII file or to change or erase some characters in it; EDI is the valid abbreviation. |
| CREATE | Use CREATE when you want to create a new ASCII file and store it on one of your diskettes; CRE is the valid abbreviation. |

Whenever MINC is READY, you can run the keypad editor with one of the three commands. The INSPECT command requires an input file name. The form of the INSPECT command is:

INSPECT filespec

Because you cannot add to or change a file that you are inspecting, the editor does not create an output file. When you finish inspecting a file, the keypad editor stops, and MINC signals READY.

The EDIT command also requires an input file name. You may specify an output file name or have MINC compose the output file name from defaults. The general form of the EDIT command is:

EDIT input-filespec output-filespec

The output file specification is optional. While you are editing, the keypad editor uses a temporary copy of your input file until you finish the editing session. When you finish, the keypad editor program creates the permanent output file and stops. MINC then signals READY. If you omit an explicit output file name, the keypad editor preserves your original input file and creates your new output file in two steps, as follows.

1. The keypad editor renames the input file type to .BAK. If a .BAK file exists with the same name as the file you are editing, MINC replaces it.
2. The keypad editor creates a new file with the input file's file name and file type.

The CREATE command requires an output file name. The form of the CREATE command is:

CREATE filespec

While you are typing the new file and correcting it, the keypad editor maintains it in a temporary form. When you finish, the keypad editor creates the new file with the name you have specified, and MINC signals READY.

If you specify a file name that already exists, the keypad editor displays the following message.

?EDITOR-W-Output file name is already in use,
do you want to erase its current contents (Y or N)?

Type N if you do not want the keypad editor to erase the existing file. The keypad editor stops immediately, and MINC signals READY. You can then complete the CRE command with a different file name.

Type Y if you do want the keypad editor to erase the existing file and store the new text you enter under the name you specified.

MINC'S KEYPAD AND OTHER SPECIAL KEYS

Figure 14 shows how the keypad editor uses the standard keys on your terminal's keypad and main keyboard for its special operations. Each keypad editor operation requires only one keystroke. Notice that the keypad in Figure 14 has the keypad label pasted on it. The keypad label shows you the functions of the keypad keys when you are using the keypad editor.

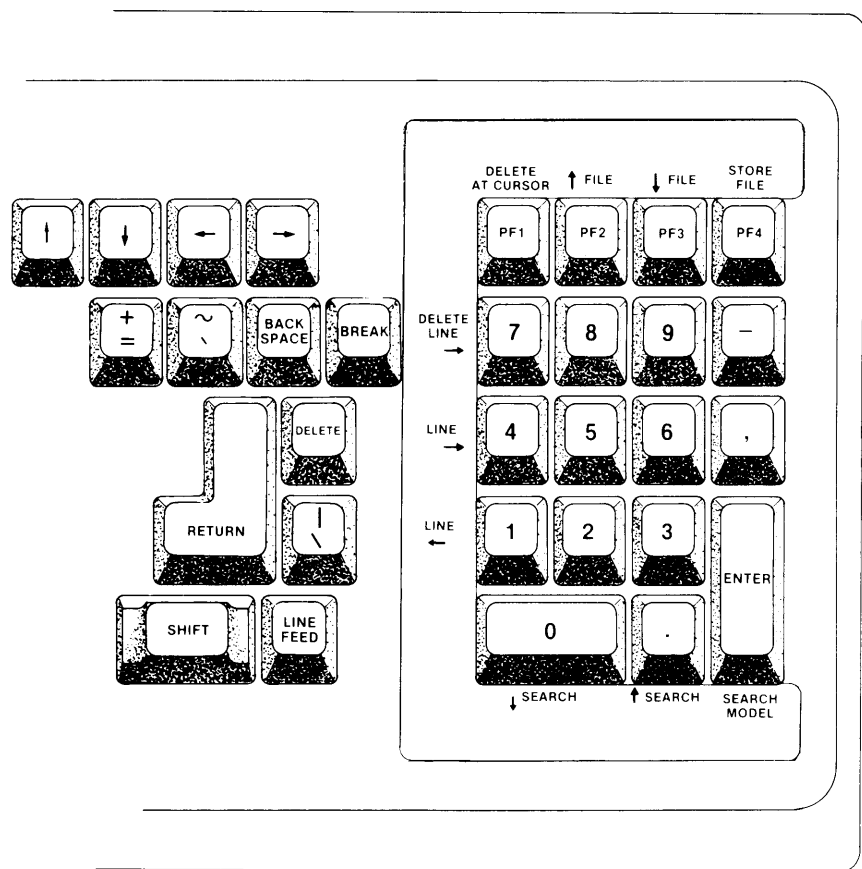


Figure 14. The Keypad

A full introduction to the MINC keypad editor follows. The first section covers the INS command and the editor operations that are active while you are only inspecting a file. The second section covers the EDI command and the additional operations that are active while you are adding to or modifying the contents of a file. The third section covers the CRE command.

Many MINC users find that the keypad editor is not particularly convenient for typing in or changing BASIC programs. On the other hand, many others use the keypad editor heavily for program entry. Consider the following major factors before you work on your own programs with the keypad editor.

BASIC Programs and the Keypad Editor

- Normally, MINC checks each program statement as you type it. It signals several sorts of significant errors immediately. The keypad editor does not check for any BASIC errors. Therefore, you really should use BASIC to create BASIC programs and not the keypad editor. See Book 3 for a more detailed discussion of this problem.
- The keypad editor provides ways to erase and insert characters that are different from the DEL and SUB commands. (You cannot use DEL or SUB while you are using the keypad editor.)
- In MINC's normal mode, your current program is always accessible in MINC's workspace. When a program doesn't quite work, you can modify it with immediate BASIC statements and, in many cases, correct it dynamically. When you use the keypad editor, you cannot run the program or use any immediate statements or commands.
- Normally, MINC displays your current program only when you use a LIST command. As you type new commands and statements, your program statements scroll upward and off screen. When you use the keypad editor, 24 lines from the file you are editing are always on the screen.

Again, the INSPECT command allows you to look at an existing ASCII file that you do not want to change in any way.

INSPECTING AN ASCII FILE

Again, the form of the INS command is:

INS filespec

If you leave out the device, MINC defaults to SY0:. If you leave out the file type, MINC defaults to .BAS.

Because you cannot add to or change a file that you are inspecting, the editor does not create an output file. When you finish inspecting a file, the keypad editor stops, and MINC signals READY.

This whole section describing the INSPECT feature of the keypad editor assumes that you are using the EDITOR.001 file from a demonstration diskette. If you do not have a copy of this diskette, install the master demonstration diskette in SY0: and an unused diskette in SY1:, and then type the RESTART command to copy the Master Demonstration diskette.

NOTE

For the steps in this section, use a demonstration diskette and install it in SY0:.

INS Operations and Symbols

This section describes the special ASCII characters and the terminal keys that are important to the keypad editor for inspecting a file. These keys are described in general here and are described again in later sections where the concepts are demonstrated in the context of actually inspecting a file.

TERMINATORS	The special ASCII characters that define the ends of lines in your file — FF, VT, CR, LF, and the CR LF combination.
STORE FILE	Return to MINC's normal mode.
CURSOR	The cursor marks the insert position and defines the upper and lower parts of your file.
↑ FILE	Move the cursor to the top of your file.
↓ FILE	Move the cursor to the bottom of your file.
↑ SEARCH	Search backward in the upper part of your file for a string that matches the current search model.
↓ SEARCH	Search downward in the lower part of your file for a string that matches the current search model.
SEARCH MODEL	Change the search model.

←	Move the cursor 1 character to the left.
→	Move the cursor 1 character to the right.
↑	Move the cursor up one line vertically. Note: this is tricky if there is no character directly above.
↓	Move the cursor down one line vertically. Note: this is tricky if there is no character directly below.
LINE →	Move the cursor to the end of the current line or, if it is at the end of a line, to the end of the following line.
LINE ←	Move the cursor to the beginning of the current line or, if it is at the beginning of a line, to the beginning of the preceding line.
⌘	End of File symbol — The crosshatch symbol appears immediately after the last character in your file.

The example for this section is the file EDITOR.001, a blank calendar for June, 1980.

When MINC displays READY, type the following command.

```
INS EDITOR.001
```

Most of the calendar presented in Figure 15 should appear on your screen. However, because the entire calendar does not fit on your terminal screen, you will see only to row s.

The INS command displays any ASCII file. You can scan the file, but the operations to insert or erase characters are inactive. The only valid operations are the ones that move the cursor. By moving the cursor through a file that is longer than 24 lines, you can display different 24-line sections of it.

```
//345678This is a sample file for MINC Keypad Editor exercises. //  
//'    '        '      Do not erase it.'          '
```

JUNE, 1980

	SUN	MON	TUE	WED	THU	FRI	SAT
a							
b							
c	1	2	3	4	5	6	7
d							
e							
f							
g							
h	8	9	10	11	12	13	14
i							
j							
k							
l	15	16	17	18	19	20	21
m							
n							
o							
p	22	23	24	25	26	27	28
q							
r							
s							
t	29	30					
u							
v							
w							
x							
y							
z							

Figure 15. The Calendar in File Editor.001

The following steps are the foundation for all later keypad editor exercises.

INS Exercises

This section shows you how to use the keypad editor to inspect a file. The procedure for inspecting a file is shown through a set of exercises that inspect the file EDITOR.001.

Each exercise is denoted by an exercise number and a title. The number is given for quick reference, and the title tells you the point of the exercise.

1. The form of the cursor.

Notice that a flashing box, the keypad editor's cursor, is at the upper left corner of your screen. The cursor always appears in this form when you are using the keypad editor.

2. The purpose of the cursor.

The purpose of the cursor is to mark your current position

on the terminal screen. The part of a file preceding the cursor at any moment is considered to be “above” it, and the part of a file following the cursor is “below” it. To use the editor’s operations correctly and efficiently, you need to be aware of the distinction. With the cursor at the upper left corner of your display, your file is entirely below it.

3. The term *current character*.

Wherever the cursor is located on your screen, the character at the cursor is the first character in the lower part of your file. That character is also called the *current character*.

The character to the cursor’s left, if any, is always the last character in the upper part of your file.

4. →

(If the cursor is not at the upper left corner of your screen, press ↑ FILE.)

Type the → key eight times.

Each right arrow operation moves the cursor one character to its right. After eight operations, *T* in the word *This* is the current character, and *8* is the last character in the upper part of your file.

5. ↓

Type the ↓ key three times.

Each downarrow operation moves the cursor vertically down one line in the file. If the next line has no character in that column (remember that spaces and tabs are characters), the cursor moves to the character at the end of that line. After those operations, the cursor should be on the *J* of *June*. Three complete lines are now in the upper part of your file. The space before the cursor is the last character in the upper part of your file.

6. ↑

Type the ↑ key three times.

Each uparrow operation moves the cursor up vertically one

line. After three uparrow operations, the cursor is back at the *T* of *This* in the top line of your screen. *T* is the current character.

7. ←

Type the ← key three times.

Each leftarrow operation moves the cursor one character to its left.

**Introducing ↑ FILE,
↓ FILE and Tone**

In the preceding steps you moved the cursor away from the top of your file and back to the top. The following steps:

- illustrate the ↑ FILE and ↓ FILE operations
- introduce the keypad editor's warning tone

8. ↓ FILE.

Type the ↓ FILE key.

↓ FILE moves the cursor to a position below your entire file. The entire file is now above the cursor.

9. The End of File Symbol.

Move the cursor to the beginning of line *w*. (The easiest way is with the uparrow key.)

Note that the special graphic crosshatch is the last character on your screen. The keypad editor displays this symbol immediately after the last character in your file. The symbol means "End of File" and it is not a character in the file the editor is displaying. As you see it in this step, the symbol shows that the last real character in the file EDITOR.001 is the terminator at the end of line *w*.

10. The warning tone.

Type downarrow to move the cursor back to the bottom of your file, and then type ↓ FILE.

Note that the cursor does not move; the editor signals that the operation failed by sounding the tone on your terminal.

The keypad editor uses the tone to signal every operation

that fails. No printed messages appear while you are using the keypad editor. Occasionally, an error message appears at the beginning or end of a session.

In most cases, the cause of the warning will be clear when you look at your screen. When you hear the tone, that means that you have tried to perform an invalid operation. The keypad editor does not do anything but sound the tone in this case — you have not hurt or destroyed your file.

11. ↑ FILE.

With the cursor at the bottom of your file, type the ↑ FILE key.

↑ FILE moves the cursor to the top of your file. The cursor always moves to the upper left corner of your screen, and the first character in your file becomes the current character.

12. More work with ↑ FILE.

With the cursor at the *J* in *JUNE* or somewhere else in the middle of a line, type ↑ FILE again. Note that ↑ FILE always moves the cursor to the top of your file from wherever it is.

13. The tone again.

Type ↑ FILE again. In this case, the tone is a signal that ↑ FILE is invalid because the cursor is already at the top of your file.

14. Practice with ↓ FILE.

Move the cursor to a place anywhere within your file; then type ↓ FILE. Note that the cursor moves to the bottom of your file from wherever it is.

The arrow operations are the most elementary operations the keypad editor provides. However, you are less likely to use them while you are inspecting a file than during an EDI or CRE session. They are most useful for moving the cursor to particular positions on your screen to insert text or erase characters, and those operations are not active during an INS session.

Introducing Searching

The keypad editor also has three searching operations, and they

are extremely useful while you are inspecting a file.

The steps to follow when you “search” for a string in your file are:

- a. Type the SEARCH MODEL key.
- b. Type in the string you want the editor to find.
- c. Specify whether the editor is to search the upper part of your file or the lower part of your file.

The string you type is the *search model* the editor uses. The upper part of your file includes all of the characters to the left of the cursor and above it. The lower part of the file includes the current character and the characters to the right of the cursor and below it.

When you have typed a search model and specified the part of your file to search, the editor moves away from the cursor and stops when it finds a string of characters in your file that matches the string you typed as a model.

The following steps:

- Demonstrate how to enter a search model.
- Introduce the concept of a cursor target.
- Illustrate the ↑ SEARCH and ↓ SEARCH operations.
- Demonstrate two common search failures.

15. Entering a search model.

Move the cursor to the top of your file. Type the SEARCH MODEL key. (Note: you do not have to move the cursor to the top of the file to use the SEARCH MODEL key. This step makes this exercise easier to describe.)

Use the SEARCH MODEL operation each time you want to specify a new model for searching operations. The editor temporarily erases the first two lines. You can specify any model up to 40 characters long, and you can use any characters on your main keyboard. A search model cannot include any characters from keypad keys.

16. Terminating a search model.

(Do not press RETURN.)

Type the single digit 2 as the model for this step, and then type the ↓ SEARCH key.

There are three ways to terminate a search model. If you change your mind about searching for anything, type CTRL/U. CTRL/U erases any partial model you have typed, cancels the SEARCH MODEL operation entirely, and leaves the cursor where it was when you typed SEARCH MODEL.

The two more common ways to terminate a search model, however, are the ↓ SEARCH and ↑ SEARCH operations.

17. ↓ SEARCH.

The ↓ SEARCH operation in step 16 moved the cursor to the 2 in the space for *June 2, 1980*. That 2 becomes the current character. The upper part of your file includes the first eight complete lines and the part of line *d* to the left of the cursor.

Each ↓ SEARCH operation moves the cursor through the lower part of your file. The search is successful when the editor finds a string in the file that matches the current model. A successful search stops with the cursor on the first character of a string that matches the model.

18. Repeated searching.

With the cursor on the 2 in line *d*, type ↓ SEARCH twice.

The editor always recalls the last model you specified and uses it until you specify a different model. This step moves the cursor to the 2 in 20, the second occurrence of that digit below the 2 in line *d*.

You can use as many ↓ SEARCH and ↑ SEARCH operations as you like for each model you specify.

19. ↑ SEARCH.

Without entering a model, type ↑ SEARCH while the cursor is at 20.

Each successful `↑SEARCH` operation moves the cursor through the upper part of your file to the first string that matches the current model. The 2 in 12 becomes the current character.

20. Search failures.

With the cursor at 12, type `↑SEARCH` twice.

The first search upwards for 2 is successful; the 2 in line *d* becomes the current character. However, the second operation fails. Because it was an `↑SEARCH` operation, the cursor moves through the upper part of your file and stops at the top.

Each time a search operation fails, the cursor moves to the top or bottom of your file and the editor sounds the tone on your terminal. Unsuccessful `↑SEARCH` operations move the cursor to the top. Unsuccessful `↓SEARCH` operations move the cursor to the bottom.

21. The concept of target.

Later steps use the term target to refer to the position where the cursor stops when the editor finishes an operation. For example:

- The target of a successful `↑SEARCH` operation is the first character of the nearest string in the upper part of your file that matches the current model.
- The target of a successful `↓SEARCH` operation is the first character of the nearest matching string below the cursor.
- The target of any unsuccessful search operation is the far end of your file — either the top or the bottom.

Introducing Unique Search Models

Searching is the fastest way to move the cursor to a precise position in your file. The following steps demonstrate two techniques for choosing search models that may not immediately be obvious:

- seeing unique combinations
- using invisible characters in models

22. Seeing unique combinations.

In the preceding searching exercises, 2 has been our target. If you type several ↓ SEARCH operations, the cursor moves from the top of your file to the bottom, stopping in the boxes for June 2, June 12 and each of the dates from June 20 through June 29. When a ↓ SEARCH operation for 2 finally fails, the cursor moves to the bottom of your file and the tone sounds.

That is a tedious way to reach the twelfth 2 in your file or any other character that is both common and far from the cursor. What is the shortest model for a ↓ SEARCH operation that moves the cursor directly from the top to the vertical bar before 20?

With the cursor at the top of your file, enter the search model |2 and search forward.

The vertical bar in |20 becomes the current character. The other strings above |20 do not match the model.

23. Another unique combination.

How can you move the cursor directly to the vertical bar to the right of 30? With the cursor at |20, try using the model 0 Ⓢ Ⓢ Ⓢ Ⓢ |, where Ⓢ represents the space character. (Enter the model and search down.)

That model is almost adequate — but not quite. The cursor moves to the 0 in |20. Another ↓ SEARCH operation moves the cursor into the June 30 block under the 0 in 30, closer to the goal, but still not quite reaching it.

24. Aha!

There is a way! Now search downward for | Ⓢ Ⓢ.

25. Using a line terminator in models.

Move the cursor to the bottom of your file, and search up for the T in SAT. Enter the model T Ⓢ and search up.

Each time you type the RETURN key when you are working with MINC, your terminal sends the two usually invisible characters CR and LF, which stand for carriage return and line feed. When a program you are using creates a

sequential file (by using PRINT statements to a file that is open for output), it also stores a CR LF pair at the end of each line. The CR LF pair is the most common line terminator, by far, because CR and LF are the characters MINC uses to represent the RETURN key. Although these characters do not always appear when you are editing a file, the characters are real, and you can use them in search models.

Each time you type the RETURN key while you are completing a search model, the editor displays special graphics for CR and LF. In this step, the model appears as *T CR LF*, and it specifies a T that is the last character in a line.

26. Using the TAB key in models.

CR and LF are the most common, usually invisible characters in ASCII files, and they appear together most of the time. The other common invisible character is HT, the character your terminal sends when you type the TAB key.

The calendar for June, 1980 includes tabs. Find the first one by moving the cursor to the top of your file, typing the TAB key once as your search model and searching downward. Each time you type the TAB key in a search model, the editor displays the special graphic character for HT.

The cursor moves just to the right of *//* at the beginning of line 2 in your file. The HT character in that position becomes the current character. It is the fourth character in its line.

27. More HT characters

With the cursor at the first HT character in line 2, type
↓ SEARCH once.

The cursor moves to the right of the second apostrophe in line 2.

Now type ← twice and then → twice. The cursor moves to column 3 and back to the right of the second apostrophe.

In fact, the character between apostrophes is HT. The first of the two HT characters is in column 4; the second is in column 10. The following step explains how the keypad editor composes your screen's display when your file has HT characters in it.

28. Explanation of tabs.

On a normal typewriter, the tab key creates a blank. The number of print columns in the blank is exactly the number of print columns before the next tab you have set with the TAB SET key.

MINC's keypad editor displays HT characters in almost the same way. MINC sets tabs automatically every eight columns — in columns 9, 17, 25, 33, and so on. (You cannot override these settings.) When the keypad editor displays an HT character, it leaves a blank to the right of the HT character and up to the next tab.

Step 26 demonstrates how the keypad editor displays HT characters. In line 2, column 4, is an HT character. The next tab is set at column 10. Column 4 is blank because HT is an invisible character. Columns 5, 6, 7 and 8 are blank because of the effect of the preceding HT character.

Step 27 demonstrates that the blank from column 4 to column 8 has only one real character in it. Therefore, the → and ← operations move the cursor one real character to the left and right, but in your display the cursor jumps across the entire blank.

Tabs are almost always confusing, but sometimes they may be necessary for the work you will be doing with ASCII files on your MINC system. The most important facts about tabs are the following.

- The special character HT represents each tab in the search model.
- HT is a single character. When a ← or → operation moves the cursor more than one screen position to the left or right through a blank, a tab is present.
- The keypad editor does not display any HT characters that are in a file you are inspecting, editing or creating. However, if you want to search for an HT character, the editor displays the special graphic for HT each time you type the TAB key as part of a search model.
- If nothing requires you to put tabs into a file, avoid using them — at least until you are comfortable with them.

- The principal benefit of using tabs is that they use much less file space than an equivalent number of spaces, but in most applications file space is not critical enough to justify using them.

The two remaining cursor movement operations in the keypad editor are `LINE←` and `LINE→`. The following steps demonstrate them.

29. `LINE→`.

Move the cursor to the top of your file and type `LINE→`.

When the cursor is not at the end of a line, the `LINE→` operation moves the cursor to the line terminator of its current line. In this step the cursor moves to the end of line 1. The terminator that follows `//` becomes the current character.

NOTE

At this point, the cursor is on the CR of the CR LF pair at the end of the line. However, nothing shows. If you type `→`, the cursor moves to the beginning of the same line and displays CR because the current character is now the LF of the CR LF pair.

30. The other function of `LINE→`.

Type `LINE→` three more times.

The cursor moves to the ends of lines 2, 3 and 4. The terminator at the end of line 4 becomes the current character. Whenever the current character is a line terminator, `LINE→` moves the cursor to the end of the following line.

31. `LINE←`.

Search downward for 20, and then type `LINE←`.

When the cursor is not at the beginning of a line, the `LINE←` operation moves the cursor to the beginning of the current line. In this step, the 1 at the beginning of line 1 becomes the current character.

32. The other function of `LINE←`.

With the cursor at the beginning of line 1, type `LINE←` three more times.

In each case, the cursor moves to the beginning of the preceding line. Whenever the cursor is at the beginning of a line, the `LINE←` operation moves the cursor to the beginning of the preceding line.

33. Finishing an INS session.

Step 32 completes the introduction to the operations that are active while you are inspecting a file. The essential step that has not been covered yet is how to finish an INS session.

Press the `STORE FILE` key.

The `STORE FILE` operation is the normal way to finish any session with the keypad editor.

When you press `STORE FILE`, MINC displays its `READY` message. If you have a program in your workspace when you begin to inspect, edit or create a file, that program is still there after you press `STORE FILE`.

The `EDIT` command allows you to add to an existing ASCII file or to change or erase some of the characters in it.

EDITING AN ASCII FILE

Again the form of the `EDIT` command is:

`EDIT input-filespec output-filespec`

For the `input-filespec`, if you leave out the device, MINC defaults to `SY0:`. If you leave out the file type, MINC defaults to `.BAS`.

For the `output-filespec`, if you leave out the device, MINC defaults to the `input-filespec` device. If you leave out the name, MINC defaults to the `input-filespec` name. If you leave out the file type, MINC defaults to the `input-filespec` file type and changes it to `.BAK` on the input file.

There are restrictions on defaulting parts of the `output-filespec`. For a description of these restrictions, see Book 3. For now, you are safest defaulting the entire `output-filespec` (leaving it out entirely) or completely specifying the entire `output-filespec`.

When you add to or change a file with the `EDIT` command, MINC creates a temporary file that holds the input file.

This whole section describing the EDIT feature of the keypad editor assumes that you are using the EDITOR.002 file from a demonstration diskette. If you do not have a copy of this diskette, install the Master Demonstration diskette in SY0: and an unused diskette in SY1:, and then use the RESTART command to copy the Master Demonstration diskette.

NOTE

For the steps in this section use a demonstration diskette and install it in SY0:.

The keypad editor works equally well when your terminal's screen width is 80 columns or 132 columns. However, for the following instructional steps, please use the 80 column width. The instructions are not accurate for 132 column displays. See Book 7 for detailed instructions on changing your screen width.

EDI Operations and Symbols

All of the operations, symbols, and concepts covered in the preceding section (the INS command) also apply to the EDIT command. See the summary of INS operations and terms preceding step 1. The following definitions describe the special ASCII characters and terminal keys that are important to the keypad editor for editing a file. These characters and concepts do not apply to inspecting a file. These concepts are described in general here and are described again in later sections where the concepts are demonstrated in the context of actually editing a file.

INSERTION	Each character you type on your main keyboard is inserted at the current character's position; the cursor, the current character, and all other characters in the lower part of your file all move to the right.
◆Ⓢ	The keypad editor displays this 2-character wrap symbol in columns 1 and 2 whenever a line is longer than 78 characters.
DELETE	Erase the character at the cursor's left.
DELETE AT CURSOR	Erase the current character.
CTRL/U	Erase all characters between the preceding line terminator and the cursor.

DELETE Erase the current character, any characters that are to its right on the current line, and the current line's terminator.

LINE→

The example for this section is the file EDITOR.002, a faulty draft of a calendar for July, 1980.

For this section, you will need to store the corrected form of the calendar under a file name you choose. When MINC is READY, type the following command.

EDI EDITOR.002 your-file-name

When you type this command, a portion of the calendar shown in Figure 16 will appear on you terminal screen.

```

      1      2      3      4      5      6      7
1234567890123456789012345678901234567890123456789012345678
♦ 90
//345678This is a sample file for MINC Keypad Exercises. //
//          ' Do not erase it.'          //

JY, 1980
aSAT
b
c  +-----+
d  |         | 1   | 3   | 4   | 5   |
e  |         |    |    |    |    |
f  |         |    |    |    |    |
g  +-----+
h  | 6   | 17  | 8   | 9   | 10  | 11  | 12  |
i  |     |    |    |    |    |    |
j  +-----+
k  |     |    |    |    |    |    | 13  | 14  |
l  ♦ 15  | 16  | 17  | 18  | 19  |    |    |
m  |     |    | *  |    |    |    |    |
n  |     |    |    |    |    |    |    |
o  +-----+
p  | 20  | 22  | 23  | 24  | 25  | 26  |
q  |     |    |    |    |    |    |
r  |     |    |    |    |    |    |
s  +-----+
t  | *  | 27  | 28  | 29  | 30  | 31  |
u  |     |    |    |    |    |    |
v  |     |    |    |    |    |    |
w  +-----+
JANUARY,
FEBRUARY,
MARCH,
APRIL,
MAY
#

```

Figure 16. The Faulty Calendar in File EDITOR.002

EDI Exercises

The easiest way to modify or correct a BASIC program is to bring it into your workspace (with the OLD command). You can then replace entire statements or use the SUB command to make substitutions within existing statements. That procedure does not work for files that are not programs, however, and for large programs it may be inconvenient. MINC's keypad editor is the most convenient tool for any large editing task. The keypad editor offers the only practical way to create, modify, and inspect files that are not programs.

This section introduces and explains the five editing operations you can use to change the contents of an ASCII file. Each step makes heavy use of the keypad editor's cursor movement operations. If you want to review the cursor movement operations, refer to the summaries at the beginning of the preceding section (the INS command).

Each time you begin an EDI session, the keypad editor displays the first 24 lines of your file. The cursor is in the upper left corner of your screen. The first character in your file is the current character, and your entire file is below the cursor.

Throughout the session, the cursor always marks the current character, except in two specific cases. The most common exception is when the cursor is at the bottom of your file, beyond the last real character in your file. The second exception is in the rare case that you move the cursor so that a CR character is immediately to its left. One of the steps below covers this case in detail. In general, all characters to the cursor's left and above it are in the upper part of your file.

34. Recognizing wrapped lines.

The first two lines in EDITOR.002 number the columns on your screen and were used when the file was created. The first correction to the file is to remove them, but before you do that, study them for a moment and finish reading this step.

When your terminal is set for an 80 column screen width, the keypad editor uses only 78 columns. When a line in a file you are inspecting, editing, or creating is longer than 78 columns, the keypad editor automatically "wraps" the line onto the following line of your display. The special wrap symbol (diamond) marks each wrapped line, and it means that the preceding line on your screen does not end with a line terminator.

For example, the screen display of the file EDITOR.002 shows that the second line of the file is longer than 78 characters. That line begins on the second line of your screen. The third line of your screen begins with the wrap symbol (diamond) and continues with the characters that do not fit on the line above. The wrap symbol means that there is no line terminator after the 8 in column 78 in the preceding line.

The keypad editor processes wrapped lines and lines that are 78 characters long (or shorter) in exactly the same way. You can demonstrate this quickly by moving the cursor to the top of your file and typing LINE→ twice. The LINE→ operation moves the cursor to the right until it reaches a line terminator. The first LINE→ operation moves the cursor to the right of the 7 in line 1 of your screen. The second LINE→ operation shows that the second line terminator in your file is displayed on the third line of your screen.

NOTE

When your terminal is set for 132 columns, the keypad editor uses 130 columns and wraps lines that are longer than that. To demonstrate this, however, you must set the terminal screen width before you run the keypad editor with the INSPECT command, EDIT command, or CREATE command. To set the screen width, see Book 7.

35. DELETE LINE→.

Move the cursor to the top of your file and type the DELETE LINE→ key.

The keypad editor erases the first line of your file, including its terminator. The first character of the second line becomes the current character, and the editor smoothly scrolls your file upward in order to show you 24 lines of the file.

Each DELETE LINE→ operation erases the current character, any characters to its right on the same line, and the line's terminator.

36. Erasing a wrapped line.

Type DELETE LINE→ once more.

The keypad editor erases the overlong line in your file. Note that it erases all of the characters up to and including the wrapped line's terminator.

37. Inserting new characters.

Move the cursor to the *Y* in *JY* on line 3. (One way is to search downward for *Y*.) Type *UL*.

The letter *Y*, the cursor, and the characters to the right of the cursor move to the right as you insert each letter. The keypad editor displays each letter immediately. The new letters are inserted before the current character.

38. Inserting new lines.

Move the cursor to *a* in line *a*. (With the cursor at the top of your file, using the model *asa* and the ↓ SEARCH operation will move the cursor to this target.) With *a* as the current character, type RETURN once.

During an EDI session, each key on your main keyboard inserts its character immediately at the cursor's position. The editor displays the character immediately. The cursor, the current character, and all other characters in the lower part of your file move one position to the right.

The *a* in line *a* is still the current character, but there is now one blank line before line *a*.

39. Correcting typing errors with DELETE.

Move the cursor to the *S* in *SAT*. (The → is the easiest way to do this.) Type 123, and then type the DELETE key three times.

NOTE

This is the DELETE key next to the RETURN key. The DELETE AT CURSOR key works differently.

You can immediately erase any typing mistake you make while you are editing or creating a file by using the DELETE key. Each DELETE operation erases the last character in the upper part of your file — the character just to the left of the cursor. The DELETE key works the same in the editor as it usually does.

40. Inserting new characters.

With the cursor on the *S* in *SAT*, complete the calendar's banner. Type the following line, inserting spaces where indicated (do not include parentheses).

```
(9 spaces)SUN(4 spaces)MON(4 spaces)
TUE(4 spaces)WED(4 spaces)THU(4 spaces)FRI(4 spaces)
```

SAT moves to the right as you insert each character. Lines below line *a* do not change because the insertion does not make line *a* wider than your screen.

The only time insertion is invalid is when your file is too full to accept another character.

41. Inserting another character.

The 2 for *July 2* is missing from its box. Move the cursor to the space in line *d* that is under the *W* in *WED*. (Searching forward for | Ⓢ 3 is one way to reach that target.) Insert a 2.

The characters to the left of the cursor do not move, but the new 2 aligns the right part of the line properly.

42. DELETE again.

Move the cursor to the 7 in line *h*. (Searching downward for 17 and using → is one way. Searching downward for 7 is quicker.) Type DELETE once.

The editor removes the character 1 and closes up the right part of the cursor's line from the right. After erasing the 1 in line *h*, the rest of the line is aligned properly.

The only times DELETE is invalid are when the cursor is at the top of your file or when there are no characters in your file.

WARNING

Avoid using the DELETE operation to erase line terminators until you have extensive experience with the keypad editor. Later steps in this section demonstrate the kind of graphic confusion that can occur when you use DELETE to erase terminators and show in detail how other editing methods are less confusing.

43. Inserting a RETURN to break a line.

Study line k briefly. It is too long to fit in one screen line because line l is joined to it. The wrap symbol (diamond) appears below k to show that the line is too long. To fix this fault in the calendar, you need to insert a terminator after the last + in line k and then identify and align line l .

Move the cursor to the space after the last + in line k (search downward for +). Press the RETURN key.

This step inserts a CR LF pair between the + and the space following it. Each time you insert a RETURN between two characters, the current character becomes the first character of the new line.

44. Another insertion exercise.

Insert an l at the beginning of line l to align it properly.

45. DELETE AT CURSOR.

Move the cursor to the first * in line o and insert six hyphens (-). After inserting the characters, type DELETE AT CURSOR six times.

Each DELETE AT CURSOR operation erases the current character and the next character becomes the current character. The editor closes up the line from the right if any printing characters remain on it.

The only times DELETE AT CURSOR is invalid are when the cursor is at the bottom of your file or when there are not any characters in your file.

WARNING

Avoid using the DELETE AT CURSOR operation to erase line terminators until you have extensive experience with the keypad editor. Later steps in this section demonstrate the kind of graphic confusion that can occur when you use DELETE AT CURSOR to erase terminators and show in detail how other editing methods are less confusing.

46. DELETE LINE→ again.

Either line *s* or the line that follows it must be erased. Since line *s* is already aligned properly, this step shows how to remove the line below line *s*.

Move the cursor to the beginning of line between line *s* and line *t*. (The fastest way is with five ↓ operations.) Type DELETE LINE→.

With the cursor at the beginning of a line, each DELETE LINE → operation erases the following line.

The only times DELETE LINE→ is invalid are when the cursor is at the bottom of your file or when there are not any characters in your file.

47. Editing with the CTRL/U operation.

The fault in line *t* is that there are too many characters to the left of | 27. Move the cursor to the space at the right of the * in line *t*, and type CTRL/U.

When the cursor is not at the beginning of a line, the CTRL/U operation erases all of the characters between the cursor and the beginning of its current line. The current character remains the same, but it moves to the beginning of the line, along with characters to its right.

Now type in the *t* to align the line.

48. Erasing an entire line with CTRL/U.

Move the cursor to the beginning of line *w*, and insert the following sentence.

“This is the end of EDITOR.002.(RET)”

This step demonstrates how you can easily erase an entire line after you have typed the RETURN key. With the cursor at the beginning of line *w*, type CTRL/U.

When the cursor is at the beginning of a line, the CTRL/U operation erases the entire preceding line. When you type several CTRL/U operations, the keypad editor erases lines in the upper part of your file. When you type several DELETE LINE→ operations, the keypad editor erases lines in the lower part of your file.

49. Joining separate lines.

The last lines in your file are there to demonstrate the least confusing way to join two separate lines in your file. In this short series of steps you will also see why using the DELETE AT CURSOR and DELETE operations to erase a line terminator is usually confusing.

Move the cursor to the end of the line that has the word *FEBRUARY*. (Search forward for *FEB* and use a LINE→ operation to ensure that the cursor is at the end of the line.) Press DELETE LINE→.

The least confusing way to join two lines is to move the cursor to the end of the upper one and erase that line's terminator with a DELETE LINE→ operation. Note that the keypad editor processes DELETE LINE→ in the following straightforward way.

- It tests the current character to see if it is a terminator.
- It erases the current character.
- If it has erased any terminator, it stops erasing; otherwise it repeats these three steps.

The keypad editor recognizes five line terminators — FF, VT, LF, CR and the special (but most common) case of the CR LF combination. DELETE LINE→ is the least confusing way to join lines because the editor handles all five line terminators equally well when you use DELETE LINE →.

REMINDER

DELETE LINE→	Deletes a line from the current cursor position to the next terminator.
CTRL/U	Deletes from before the cursor to the preceding terminator.

Move the cursor to the end of the line *FEBRUARY, MARCH*., Type DELETE AT CURSOR. The terminator on the line was a CR LF combination. DELETE AT CURSOR erases the CR character. The result shows how the keypad editor displays a LF character that stands alone. Type DELETE AT CURSOR again.

As you can see, a single DELETE LINE→ operation would have joined the lines in the same way.

Now move the cursor to the *M* in *MAY*. Prepare for a small surprise — and type DELETE.

The DELETE operation erased the LF character from a CR LF combination. A CR character now stands alone. In your file the string of characters around CR is PRIL, Ⓢ CR MAY.

When the editor displays the solitary CR character, it then displays the word MAY over the first three characters of FEBRUARY. The reasons for this are sound, but they are beyond the scope of this book. However, to signal what has happened in this very confusing situation, the keypad editor displays the special CR symbol in column 1. That symbol means that a stand-alone CR is at the cursor's left and the display is at least somewhat unreadable.

Type DELETE again.

Whenever you accidentally erase a LF character, immediately use the DELETE operation if the CR symbol appears on your screen.

Avoid using the DELETE AT CURSOR and DELETE operations to erase line terminators at least until you have extensive experience with editing.

50. Joining two lines — the last exercise!

For the last step in this series, move the cursor to the end of line w. Type DELETE LINE→. You can now finish fixing this file at your leisure.

51. Finishing an EDI session.

Press the STORE FILE key at this point.

The STORE FILE operation is the normal way to finish any session with the keypad editor. Until you type STORE FILE, the changes you make to a file are temporary.

This section covers the six control characters that are most important while you are inspecting, editing, or creating a file with the keypad editor.

WARNING

The keypad editor processes most control characters without causing any confusion. However, there are several control characters that can cause the keypad editor to produce a confusing or unreadable display. Therefore, avoid using any control characters while you are inspecting, editing, or creating a file *until* you are sure you want and need them.

The CTRL/U character is the only one that you will need frequently while working with the keypad editor. CTRL/U is the keypad editor operation for erasing the characters on the current line from the beginning of the line up to the cursor. Detailed instructions about inspecting, editing, and creating ASCII files appear in the instruction sections, which follow this introductory material.

The CTRL/C character is especially important while you are editing or creating an ASCII file with the keypad editor. When you type CTRL/C, the editor will discard the temporary file it was using and stop immediately, and MINC will signal READY if you answer yes to the following question that the editor displays.

?EDITOR-W-Abort edit session losing all edits (Y,N)?

If you type Y, MINC returns to the READY, changes no files and loses all your edits. If you type N, MINC returns the editor to where you were before you typed CTRL/C.

The CTRL/L character is particularly useful if your MINC system includes a line printer. (If your system does not include a hardcopy printer, CTRL/L offers no major benefits.) When you type CTRL/L, your terminal sends the ASCII character FORMFEED to MINC. If you are editing or creating a file with the keypad editor, the editor displays the FORMFEED character with the special graphic FF and processes it like any of the other valid ASCII characters. If your hardcopy printer starts a new page whenever MINC sends a FORMFEED character, CTRL/L offers the easiest way to divide an ASCII file into pages.

The CTRL/W character is important because the keypad editor interprets it as a command to update your screen display. Type CTRL/W whenever you want to be especially sure that the characters you see are exactly the ones in the part of your file you are working with. The most common case in which CTRL/W is

useful is when you have accidentally typed another disrupting control character and need to be sure that your screen is up to date.

The CTRL/Q character is important because it cancels the confusing effects of the CTRL/S character. Avoid typing CTRL/S; if you type the combination while you are using the keypad editor, the editor will continue to run but it will stop displaying what it is doing. If you do type CTRL/S by mistake, type CTRL/Q to establish immediate screen updating again. Remember that the NO SCROLL key performs the same function as the CTRL/Q and CTRL/S pair.

See Book 3 or Book 7 for detailed instructions about changing the number of columns MINC displays on your screen.

Screen Width and the Keypad Editor

The keypad editor works equally well when you set your terminal's screen for 80 columns or for 132 columns. If you want to change your screen width as you are editing, use the SET-UP MODE to change the width and then press CTRL/W.

MINC provides a distinct command, CREATE, for your use when you want to type in an entirely new ASCII file. The principal difference between the CREATE command and the EDI command is that you can only specify one file name for CREATE, while one or two file names are both valid for EDIT.

CREATING AN ASCII FILE

Again, the form of the CREATE command is as follows.

CRE input-filespec

If you leave out the device, MINC assumes SY0:. If you leave out the file type, MINC defaults to .BAS.

When you create a file with the CRE command, MINC creates a temporary file that holds your input. MINC creates the file on the diskette when you terminate the session by pressing the STORE FILE key.

You cannot create a new file using the EDI command. When you type:

EDI new-filespec

MINC prints the following message:

?EDITOR-F-Cannot find input file on specified or default volume

CRE Operations and Symbols

The operations, symbols, and concepts for a CRE session are exactly the same as for an EDI session. All of the editor's operations are active while you are creating a new file. The common exception is at the beginning of each CRE session, when there are not any characters in your new file yet. The first operation in a CRE session must be to insert one character or more.

CRE Exercises

The two preceding sections (INS and EDI) provide practice with the keypad editor. At this point you can create a file to try the CRE command.

Try typing the mailing list using the CRE command rather than the FILEMT.BAS program listed in Chapter 14.

CHAPTER 16

DEBUGGING YOUR PROGRAMS IN THE IMMEDIATE MODE

You can use the immediate mode along with the program mode to help find the errors (*bugs*) in a program. The term for finding the errors in a program is *debugging*.

When a program does not work as you anticipated it would, you can stop the program in the middle of its execution by pressing CTRL/C twice (once at an input prompt) or wait for it to terminate normally. In either case, MINC will display READY. At READY, you can use the immediate mode to print the values of variables used by the program, change the value of a variable, or restart the program from any line.

The following example shows how you might use the immediate mode to figure out what is wrong with a program.

The following program is a trivial example of a procedure you might follow with a data collection or data analysis program. This program dimensions array A at 1000. The FOR loop that processes array A, however, does nothing more than set A(I) to I in this example.

```
10 DIM A(1000)
20 FOR I=1 TO 10000
30 A(I)=I
40 NEXT I
50 PRINT 'this is the end'
60 END
```

When you run this program, you get the following result.

```
NONAME          01-JUN-80          10:06:22
```

```
?MINC-F-Array subscript is negative or too large at line 30
```

```
READY
```

Now you decide that you must look at line 30 to see what you did wrong.

```
LISTNH 30
30 A(I) = I
```

Line 30 looks fine. So the next step is to see what the value of I is. Since all arrays and variables maintain their values until you perform a SCR, RUN, or their equivalents, you can print the value of I in the immediate mode.

```
PRINT I
1001
```

The value of I is 1001. The only line that changes the value of I is statement 20.

```
LISTNH 20
20 FOR I = 1 TO 10000
```

There is a typographical error in statement 20. The high value of 10,000 was typed instead of 1000. Now you can correct the line and try the program again.

You can examine the array in the immediate mode. The following FOR loop prints out the first 20 elements of array A.

```
READY
FOR I = 1 TO 20 \ PRINT A(I); \ NEXT I
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

You can also examine an individual array element. For example:

```
READY
PRINT A(1000)
1000
```

As another example, look at the lower-to-upper-case program (not subroutine). Below is a listing for the program.

LIST

L2U 05-JUN-80 10:07:49

```

10 REM - This program converts lower case string input
20 REM - to upper case.
30 REM - If the string is lower case, it is converted to upper
40 REM - case. If the string is upper case, the program leaves it alone.
50 U = ASC('a')-ASC('A')
60 PRINT 'Input a string'; \ INPUT S$
70 FOR I = 1 TO LEN(S$)
80 T$ = SEG$(S$,I,I)
90 IF ASC(T$) >= ASC('a') THEN T$ = CHR$(ASC(T$)-U)
100 R$ = R$&T$
110 NEXT I
120 PRINT R$

```

READY

Now suppose you run this program as follows:

RUN

L2U 05-JUN-80 10:09:46

```

Input a string? {This is a string within braces.}
[THIS IS A STRING WITHIN BRACES.]

```

READY

Notice that the program has changed the braces ({}) to brackets ([]).

The first thing that you would probably do is make sure that the program input the string properly. In the immediate mode, print out the value of S\$.

```

READY
PRINT S$
{This is a string within braces.}

```

READY

As you can see, S\$ is correct.

Statement 90 is the statement that actually changes the input string.

```

LISTNH 90
90 IF ASC(T$) >= ASC('a') THEN T$ = CHR$(ASC(T$)-U)

```

This line tests to see if the input character is greater than or equal to 'a', but it does not test to see if the input character is less than or equal to 'z'. Thus, those characters that have an ASCII value greater than 'z' get converted too.

You can change line 20 and then start executing the program from the beginning of the loop at statement 70.

```
SUB 90[THEN[THEN IF ASC(T$)<=ASC('z') THEN
90 IF ASC(T$)>=ASC('a') THEN IF ASC(T$)<=ASC('z') THEN T$=CHR$(ASC(T$)-U)

READY
GO TO 70
[THIS IS A STRING WITHIN BRACES.]{THIS IS A STRING WITHIN BRACES.}
```

Notice that the output string was not cleared when the program was executed from line 70. Where you use the RUN command, MINC sets all string variables to the null string and all numeric variables to 0. However, when you use the GO TO command in the immediate mode, none of the values of variables are changed. Thus, the new upper case string is concatenated to the old.

If you set the value of R\$ to the null string, you can then start the program from line 70 and get the expected results.

```
READY
R$=""

READY
GO TO 70
[THIS IS A STRING WITHIN BRACES.}

READY
```

CHAPTER 17

WHERE TO GO FROM HERE

If you have gotten to this point and have understood all of the examples presented in this book, you have a good command of the MINC BASIC programming language. You will probably not need to refer to this manual any more.

Book 3: MINC Programming Reference is designed to help you when you need BASIC reference. All of the BASIC commands and statements are described in alphabetical order and all of the information about a command or statement is described in one place.

For example, Book 2 was designed to teach BASIC. Thus, the information was presented in an order logical for learning. All of the information about PRINT statements was not presented at once because you would have gotten lost in the detail.

However, in Book 3, all of the information about PRINT statements is presented in one place. Now that you understand PRINT statements, you will want all the information in one place so that you can find what you are looking for quickly.

Book 3 also provides you with more technical information than was presented in this manual. It describes the commands and statements in more detail, providing the more detailed and more powerful options of some statements and the more detailed restrictions.

You should browse through Book 3 to familiarize yourself with the format of the manual. Look up some of the BASIC statements and commands to see some of the more technical information.

If you have a problem when you are writing a program, look in Book 3. You might find that there are restrictions for a statement that were not described in this manual.

Another manual that will be of service to you is *Book 8: MINC System Index*. Book 8 gives explanations and/or references for all of the error messages and contains the combined indexes for Books 2 through 7.

Book 4: MINC Graphic Programming, *Book 5: MINC IEEE Bus Programming*, and *Book 6: MINC Lab Module Programming* describe each of the features of the MINC system. Each of these manuals has a tutorial section in the beginning to help you learn to use the MINC routines and a reference section at the end.

Book 7: Working with MINC Devices explains how to connect the MINC modules.

APPENDIX A

ASCII CHARACTER SET

The following table shows, with the corresponding decimal codes, the 128-character ASCII (American Standard Code for Information Interchange) character set. These codes are used to store ASCII data in files and to store them internally.

You can convert an ASCII value to the corresponding string character with the CHR\$ function and can convert a string character to the corresponding ASCII value with the ASC function (see Chapter 8).

BASIC also uses the ASCII values of the characters in string comparisons (see Chapter 5).

Notice in the table that there are ASCII codes for every character — even non-printing characters. For example, the ASCII code for the space character is 32. The two ASCII codes equivalent to the RETURN key are 13 (carriage return) and 10 (line feed). Every time you press RETURN, MINC sends both characters.

This table also shows the collating sequence. The characters are ordered by their ASCII codes.

<i>ASCII Decimal Code</i>	<i>Character</i>
0	NUL (CTRL/@)
1	SOH (CTRL/A)

PROGRAMMING FUNDAMENTALS

<i>ASCII Decimal Code</i>	<i>Character</i>
2	STX (CTRL/B)
3	ETX (CTRL/C)
4	EOT (CTRL/D)
5	ENQ (CTRL/E)
6	ACK (CTRL/F)
7	BEL (CTRL/G)
8	BS (CTRL/H)
9	HT (CTRL/I or TAB)
10	LF (NEW LINE or LINE FEED)
11	VT (Vertical TAB)
12	FF (Form Feed)
13	RT (Return)
14	SO (CTRL/N)
15	SI (CTRL/O)
16	DLE (CTRL/P)
17	DC1 (CTRL/Q)
18	DC2 (CTRL/R)
19	DC3 (CTRL/S)
20	DC4 (CTRL/T)
21	NAK (CTRL/U)
22	SYN (CTRL/V)
23	ETB (CTRL/W)
24	CAN (CTRL/X)
25	EM (CTRL/Y)
26	SUB (CTRL/Z)
27	ESC (ESCAPE)
28	FS (CTRL/\)
29	GS (CTRL/])
30	RS (CTRL/^)
31	US (CTRL/)
32	SP (space bar)
33	!
34	"
35	#
36	\$
37	%
38	&
39	'
40	(
41)
42	*
43	+
44	,

<i>ASCII Decimal Code</i>	<i>Character</i>
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W

<i>ASCII Decimal Code</i>	<i>Character</i>
88	X
89	Y
90	Z
91	[
92	\
93]
94	^
95	_
96	,
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	
125	}
126	~
127	DELETE

INDEX

- ABS function, 118
 - form of, 118
- Addition, 11
- APPEND command, 185, 196-200
 - form of, 196
- Arc tangent function, 17
- Argument, 16
 - dummy, 131
- Arithmetic
 - integer, 113-115
 - mixed mode, 114-116
 - relational operator, 76-78
- Arithmetic expression, 11
- Arithmetic functions, 117-118
 - form of, 117
- Arithmetic operations, 31
- Array element, 93-94, 96, 104-106
- Arrays
 - creation of, 94-95
 - definition of, 93
 - one-dimensional, 96-98, 104-105, 110
 - two-dimensional, 96, 98-100, 105-106, 111
- ASC function, 125-126, 129-130
 - form of, 126
- ASCII code, 77, 120, 126, 129, 241
 - definition of, 77-78
- Assignment statement, 21-22, 38, 80
 - form of, 21
- ATN function, 17, 117
 - form of, 18
- Backslash, 22-23
- Bad blocks, 69-70
 - definition of, 69
- .BAS, 50
- BASIC, 3-4
- BASIC programs and keypad
 - editing, 207
- BIN function, 125
- Block
 - definition of, 55, 63
- Body of loop, 90
- Bounds, 107-109
- Bug, 235
- Calendar example
 - editing, 223-231
 - inspecting, 209-221
- Cancelling a keypad editor session, 232
- Capitalization, 3, 28, 33, 40
- Chain, 185-194

- CHAIN statement, 185-187
 - form of, 186
- Characters, 32, 120, 126-130
- CHR\$ function, 125-127, 129
 - form of, 127
- CLEAR command, 22
- CLK function, form of, 121
- CLOSE statement, 146, 150
 - form of, 149, 168
- Closing files, 146
- Collating sequence, 77-78, 241
- COLLECT command, 66
 - form of, 68
- Combining programs, 185
- Comma
 - use of, 34-35
- Command, 2
 - definition of, 3, 43
- Commands
 - APPEND, 185, 196-200
 - CLEAR, 22
 - COLLECT, 66, 68
 - COMPILE, 61
 - COPY, 72-73
 - CREATE, 204, 205-206, 233
 - DATE, 121-122
 - DEL, 44
 - DIR, 54-56, 151
 - DUP, 70, 71-72
 - EDIT, 204-205, 221-222, 224
 - INI, 68, 71-72
 - INSPECT, 204, 207, 209
 - LENGTH, 57-58
 - LIST, 27, 58
 - LISTNH, 59
 - NEW, 26, 28
 - OLD, 52-53, 146
 - REPLACE, 53-54
 - RESEQ, 47-48, 82-83, 141-142
 - RUN, 27-28, 31, 58-60
 - RUNNH, 59-61
 - SAVE, 49-52, 145
 - SCR, 22, 28
 - Scratch, 22, 28
 - SUB, 44-47
 - TIME, 121-122
 - TYPE, 56-57
 - UNSAVE, 66
 - VERIFY, 70-71
- COMMON statement, 194-196
 - form of, 194
- COMPILE command, 61
 - form of, 61
- Compile
 - definition of, 61
- Computer, 25
- Concatenation, 39
- Conditional GO TO, 82
- Control characters, 4-5
- COPY command, 72-73
 - form of, 72
- Copying files, 72-73
- Correcting lines, 26-27
- COS function, 17, 117
 - form of, 18
- Cosine function, 17
- CR, 208
- CREATE command, 204-206
 - form of, 233
- Creating files, 233
- CTRL/C, 4-5, 26
- CTRL/C key and keypad editor, 232
- CTRL/L key and keypad editor, 232
- CTRL/Q, 5
- CTRL/Q key and keypad editor, 233
- CTRL/S, 5
- CTRL/S key and keypad editor, 233
- CTRL/U key and keypad editor, 222, 229, 230, 232
- CTRL/W key and keypad editor, 232-233
- Current character, 211
- Current owner, 68-69
- Cursor, 208, 210-211
- .DAT file type, 148, 149
- DAT\$ function
 - form of, 121
- DATA statement, 173-176
 - form of, 174
- Data type, 113
- Date, 121-122

- DATE command, 121-122
 - form of, 121
- Debugging, 235-238
- DEF statement, 130-133
 - form of, 131
- DEL command, 44
 - form of, 44
- DELETE AT CURSOR key, 222, 228
- DELETE key, 2, 222, 226, 227
- DELETE LINE→ key, 223, 225, 228-229, 230
- Device, 50, 51
 - definition of, 5-6
- DIM statement, 94, 110
 - definition of, 94
- DIM # statement, 166-167
 - form of, 166
- Dimension, 94-95, 98, 107-109
 - definition of, 94
- DIR command, 54-56, 151
 - form of, 54
- Directory, 64, 66, 68
 - definition of, 54
- Diskette, 5, 61
- Division, 12-15
- Dummy argument
 - definition of, 131
- DUP command, 71-72
 - form of, 71
- Duplicating diskettes, 71-72

- E notation
 - definition of, 10-11
 - form of, 10
- EDIT command, 204-205
 - form of, 221
- Editing
 - files, 221-231
 - files for line printers, 232
- Editor
 - See Keypad Editor, 203
- EDITOR.001 example, 210
- EDITOR.002 example, 223
- Element of array, 93
- End condition, 86-87
- End of file symbol, 209, 212

- END statement, 83
- Entering search models, 214
- Erasing a wrapped line, 225-226
- Examples
 - editing calendar, 223
 - EDITOR.001, 210
 - editor.002, 223
 - file concatenation, 192
 - file input, 188
 - file list, 193
 - file maintenance, 162-163, 187-194
 - file merge, 191-192
 - file sort, 189-191
 - input sequential file, 162-163
 - inspecting calendar, 209
 - lower to upper case, 128-130, 138-140
 - merging files, 153-162
 - questionnaire, 93, 97-103, 105, 169-170
 - yes/no validation, 100, 136-140
- Execute
 - definition of, 27
- EXP function, 19, 117
 - form of, 19
- Exponential function, 19
- Exponentiation, 14, 15
- Expressions
 - arithmetic, 11
 - logical, 76
 - string, 38

- FF, 208
- Fields, 178
- File, 49-50, 66, 145-146
- File concatenation example, 192-193
- File input example, 188
- File list example, 193
- File maintenance example, 162-163, 187-194
- File merge example, 191-192
- File number, 148-151
 - definition of, 148
- File sort example, 189-191
- File specification, 50

- File type, 50
 - .DAT, 148
 - FILE.BAD, 71
 - FILEM1 program, 189
 - FILEM2 program, 189-191
 - FILEM3 program, 191-192
 - FILEM4 program, 192-193
 - FILEM5 program, 193
 - FILEMT program, 188, 193
- Files
 - copying, 72-73
 - creating, 233
 - definition of, 145
 - editing, 221-231
 - inspecting, 210-221
 - nonprogram, 51-52
 - program, 49-51, 55
 - sequential, 146-163
 - system, 62
 - virtual array, 146-147, 163-170
- Filespec, 51
 - definition of, 51
- Finishing an EDI session, 231
- Flow
 - definition of, 75
- FOR INPUT, 148, 165-166
- FOR loop, 88, 95, 103-104
- FOR OUTPUT, 148, 165-166
- FOR statement, 89-91
 - form of, 90
- Format description, 177-179
 - centered, 183
 - commas, 180
 - decimal point, 179
 - dollar sign, 180-181
 - E notation, 181
 - extended, 183
 - left-justified, 182
 - minus sign after, 180
 - number of digits, 179
 - numeric, 179-182
 - preceding asterisks, 180
 - right-justified, 182
 - string, 182-184
 - trailing minus sign, 180
- Formatted output, 177
- FORMFEED, 232
- Function, 16
- Functions
 - ABS, 118
 - arithmetic, 117-118
 - ASC, 125-126, 129-130
 - ATN, 17, 117
 - BIN, 125
 - CHR\$, 125, 127, 129
 - CLK, 121
 - COS, 17, 117
 - DAT\$, 121
 - definition of, 16
 - EXP, 19, 117
 - INT, 118
 - LEN, 122, 128
 - LOG, 19, 117
 - LOG10, 19
 - OCT, 125
 - PI, 17, 117
 - POS, 123-124
 - RND, 118-120
 - SEG\$, 122, 124-125, 128-129
 - SGN, 118, 120
 - SIN, 17, 117
 - SQR, 17, 117
 - STR\$, 125, 128
 - string, 120-125
 - system, 176
 - TAB, 35-36
 - trigonometric, 117
 - TRM\$, 122-123
 - user-defined, 130-133
 - VAL, 125, 127
- GO TO statement, 81-82
 - form of, 81
- GOSUB statement, 136-138
 - form of, 137
- Graphic routine, 143
- Horizontal tab, 218-219
- HT, 218-219
- IF statement, 75-76
 - form of, 76

- IF END # statement, 152-153, 158-159
 - form of, 152
- IF/GO TO statement, 81
 - form of, 81
- IF/THEN statement, 78-80
 - form of, 78
- Immediate mode, 2, 7, 20, 235, 236
- Infinite loop, 86
- INI command, 68, 71-72
 - form of, 68
- Initialize, 68, 71-72
 - definition of, 64, 68
- Initialized diskette
 - graphic of, 64
- Input, 2
- Input sequential file example, 162-163
- INPUT statement, 29-31, 37-39, 173
 - form of, 31
- INPUT # statement, 151
 - form of, 151
- Inserting new characters, 226
- Inserting new lines, 226
- INSPECT command, 204-205, 207, 209
 - form of, 207
- Inspecting files, 210-221
- Instrument bus routine, 143
- INT function, 118
 - form of, 118
- Integer arithmetic, 115
- Integer variable, 21, 114
 - definition of, 113

- Keypad, 206
- Keypad editing and BASIC programs, 207
- Keypad editing
 - terminating a session, 231
- Keypad editor
 - file restrictions, 204
- Keys
 - ↓, 209
 - ↓ FILE, 208, 212, 213
 - ↓ SEARCH, 215
 - ←, 209, 212
 - , 209, 211
 - ↑, 209, 211-212
 - ↑ FILE, 208, 211, 213
 - ↑ SEARCH, 208, 215-216
- CTRL/C and keypad editor, 232
- CTRL/L and keypad editor, 232
- CTRL/Q and keypad editor, 233
- CTRL/S and keypad editor, 233
- CTRL/U and keypad editor, 222, 229, 230, 232
- CTRL/W and keypad editor, 232
- DELETE, 222, 226-227
- DELETE AT CURSOR, 222, 228
- DELETE LINE→, 223, 225, 228, 230
- LINE←, 209, 220
- LINE→, 209, 220
- NO SCROLL, 233
- STORE FILE, 208, 221, 231
- TAB, 218
- KILL statement
 - form of, 170

- Lab module routines, 111, 143
- LEN function, 122, 128
 - form of, 122
- LENGTH command, 57-58
 - form of, 57
- LET, 21
- LET statement, 21, 38
- LF, 208
- Line printers
 - editing files for, 232
- Line terminator in search models, 217-218
- LINE← key, 209, 220
- LINE→ key, 209, 220
- LINPUT statement, 38
 - form of, 38
- LINPUT # statement, 151, 154, 157-162
 - form of, 152
- LIST command, 27, 58
 - form of, 58
- LISTNH command, 59

- Literal
 - numeric, 9
 - form of, 9
 - string, 32-33
- LOG function, 117
 - form of, 19
- LOG10 function
 - form of, 19
- Logarithmic function, 19
- Logical expression, 76
 - definition of, 76
 - form of, 76
- Loop, 86-88, 90-92, 100
 - definition of, 85
 - FOR, 95
- Lower case, 2
- Lower to upper case example, 128-130, 139
- Magnitude, 10
- Master volume, 62
- Merging files example, 153-161
- Mixed mode arithmetic, 114, 115-116
- Multiple statement line, 22-23
- Multiplication, 12-16
- NAME statement
 - form of, 170-171
- Name
 - string variable, 37
- Nesting
 - expressions, 15
 - FOR loops, 103-104
 - functions, 17
 - loops, 100-103
 - definition of, 101
 - subroutines, 140
- NEW command, 26, 28
 - form of, 28
- New diskette
 - graphic of, 63
- NEXT statement, 89-90, 92
 - form of, 90
- NO SCROLL key, 4, 233
- NONAME, 28
- Nonprogram file, 51
- Nonsystem volume, 66
 - definition of, 62
- Null string
 - definition of, 37
- Numeric literal
 - definition of, 9
 - form of, 13
- Numeric variable, 20-21
- OCT function, 125
- OLD command, 52-53, 146
- ON/GO TO statement, 82, 97
 - form of, 82
- ON/GOSUB statement
 - form of, 140
- One-dimensional array, 96-98, 104-105, 110
 - definition of, 96
- OPEN statement, 147-149, 165-166
 - form of, 147, 165
- Opening files, 147-148
- Output, 2, 31-32
 - formatted, 177
- OVERLAY statement, 185, 200-201
 - form of, 200
- Overriding priority, 14-16
- Owner, 68-69
- Parentheses
 - use of, 14-15
- PI function, 17, 117
- POS function, 122-124
 - form of, 123
- Precision, 16-17
- PRINT format
 - See format description
- PRINT statement, 3
 - form of, 8
- PRINT USING errors, 184
- PRINT USING statement, 177-184
- Print zones, 33-35
 - definition of, 34
- PRINT # statement, 150
 - form of, 149

Priority, 11-16, 31
 overriding, 14
 Program, 3, 29
 definition of, 25
 Program file, 49-52, 55, 64, 145
 definition of, 49
 Program flow
 definition of, 75
 Program format, 28
 Program mode, 2, 25, 29
 Program name, 26, 28, 50
 form of, 50
 Program termination, 83

Quadratic formula, 117
 QUEST program, 101-102, 169-170
 Question mark prompt, 30-31
 Questionnaire example, 93, 97-103,
 105, 169-170

Radians, 17
 RANDOMIZE statement, 119-120
 Range of magnitude, 10
 READ statement, 173-176
 form of, 173
 READY message, 2
 Real variable, 114
 definition of, 113
 Relational operator
 arithmetic, 77
 string, 77-78
 REMARK statement, 40-41
 REPLACE command, 53-54
 form of, 53
 RESEQ command, 47-48, 82-83,
 141-142
 form of, 47
 RESTORE statement
 form of, 176
 RESTORE # statement
 form of, 153
 RETURN key, 2, 4
 RETURN statement, 136-138
 form of, 137

RND function, 118-120
 form of, 119
 Rounding, 10
 Routines,
 definition of, 143
 graphic, 143
 instrument bus, 143
 lab module, 143
 Row major order, 111
 RUN command, 27-29, 31
 form of, 59-61
 Run
 definition of, 27
 RUNNH command, 60-61

SAVE command, 49-52, 145
 form of, 51
 Saving programs, 48
 Scientific notation
 See E notation, 10
 SCR command, 22, 28
 Scratch command, 22, 28
 Screen width and keypad editor, 233
 Search failures, 216
 Search model, 208, 214, 218, 220
 entering, 214
 terminating, 215
 unique, 216-217
 Searching, 213-218
 SEG\$ function, 122, 124-125,
 128-129
 form of, 124
 Semicolon
 use of, 35
 Sequential file, 145-163
 definition of, 146
 SGN function, 118
 form of, 120
 Shell sort, 189-190
 SIN function, 17, 117
 form of, 18
 Sine function, 17
 Special characters, 4-5
 Special keys, 4
 SQR function, 17, 117

- Square root function, 17
- Statement numbers, 28-29
- Statements
 - assignment, 21, 38, 80
 - CHAIN, 185
 - CLOSE, 147, 149, 150, 167-168
 - COMMON, 194-195
 - DATA, 173-176
 - DEF, 131-133
 - definition of, 3, 43
 - DIM, 94, 110
 - DIM #, 166-167
 - END, 83
 - FOR, 88-92
 - GO TO, 81
 - GOSUB, 136-138
 - IF, 75-76
 - IF END #, 152-153
 - IF/GO TO, 81
 - IF/THEN, 78-80
 - INPUT, 29-31, 37-39, 173
 - INPUT #, 151
 - KILL, 170
 - LET, 21, 38
 - LINPUT, 38-39
 - LINPUT #, 151, 154-156
 - NAME, 170-171
 - NEXT, 88-92
 - ON/GO TO, 82, 97
 - ON/GOSUB, 140
 - OPEN, 147-149, 166-167
 - OVERLAY, 185, 200
 - PRINT USING, 177-179
 - PRINT #, 149-150
 - RANDOMIZE, 119-120
 - READ, 173-176
 - REMARK, 40-41
 - RESEQ, 141-142
 - RESTORE, 176
 - RESTORE #, 153
 - RETURN, 136-138
 - STOP, 83
- STOP statement, 83
- Storage media, 5
 - See also volumes, diskette
- STORE FILE key, 208, 221, 231
- STR\$ function, 125, 127-128
 - form of, 128
- String expression, 38
- String function, 120, 122
- String literal, 32-33
 - definition of, 32
 - form of, 32-33
- String operation, 39
- String relational operator, 77-78
- String variable, 36-38
- String variable name, 37
- Structure of volumes, 63
- SUB command, 44-47
 - form of, 45
- Subroutine, 135-142
 - definition of, 135
 - nesting, 140
- Subscript, 96, 98-99, 106-110
 - definition of, 93
- Subscripted variable, 106, 109
- Substring, 124
 - definition of, 124
- Subtraction, 11
- SY0:, 50, 62, 71
 - definition of, 5
- SY1:, 62, 64, 71
 - definition of, 5
- Symbol, wrap, 222, 224-225
- System file, 65
- System function, 176
- System volume, 62, 65, 68
 - definition of, 62
- Tab, 218
- TAB function, 35-36
 - form of, 35
- TAB key, 219
 - in search models, 218
- Target, 216
- Terminating a keypad editor session, 231
- Terminating search models, 215
- Terminators, 208
- Time, 121

- TIME command, 121
 - form of, 121
- Tone, 212-213
- Trigonometric functions, 17, 117
 - form of, 117
- TRM\$ function, 122
 - form of, 123
- Two-dimensional array, 96, 98-100, 105, 111
 - definition of, 98
- Type,
 - data, 113
 - file, 50-51
- TYPE command
 - form of, 56
- Unconditional GO TO, 81
- Unique search models, 216-217
- UNSAVE command, 66
 - form of, 66
- Upper case, 2-3
- User-defined function, 130-133
- USING, PRINT, 177
- VAL function, 125, 127-128
 - form of, 127
- Variable name, 21
- Variables, 20-22, 29-31
 - definition of, 20
 - integer, 21, 113-114
 - numeric, 20-21
 - real, 113
 - string, 36-38
 - subscripted, 106, 109
- VERIFY command, 70
 - form of, 70
- Virtual array file, 145-147, 163-170
 - definition of, 146-147
- Volume id, 55, 68-69
- Volumes, 50, 55, 62-66, 68, 145-146
 - definition of, 5
 - master, 62-63
 - nonsystem, 62, 66
 - system, 65, 68
- VT, 208
- Warning tone
 - See* Tone
- Word
 - definition of, 57
- Workspace, 20, 22, 27-28, 57, 110-111
 - definition of, 4
- Wrap symbol, 222, 224
- Wrapped lines, 224
- Yes/no validation example, 136, 139-140

In MINC you perform this calculation by typing in the following BASIC command:

```
PRINT (27 + 32)*(15-8)/327
```

MINC displays the answer beneath the PRINT statement, like this:

```
PRINT (27 + 32)*(15-8)/327
1.263
```

When you use MINC as a calculator, you are using it in what is called the *immediate mode*. In the immediate mode, MINC performs your instructions as soon as you press RETURN.

The rules for using MINC as a calculator are quite simple and are explained in this chapter.

THE PRINT STATEMENT

To get MINC to display information on the terminal, you must use the PRINT statement. As you saw above, the form of the PRINT statement is:

```
PRINT expression (RET)
```

where expression represents the number or calculation that is to be printed, and (RET) represents pressing the RETURN key. The expression is printed in blue ink, because it is optional in a PRINT statement. If you omit the expression, MINC prints a blank line.

The following three examples are valid PRINT statements where 7, 23.8, and the result of $5324.7 + 78625.9$ are to be printed by each PRINT statement respectively.

```
PRINT 7
```

```
PRINT 23.8
```

```
PRINT 5324.7 + 78625.9
```

If you enter each of the above statements, MINC prints the results on the screen as follows:

```
PRINT 7
7
```

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require an immediate answer and you are under warranty, call the appropriate MINC Customer Support Center. (The MINC Customer Support Centers are listed in the MINC Newsletter.)

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____ Telephone _____

Street _____

City _____ State _____ Zip Code _____
or Country