MINC–11

Book 2:
MINC Programming Fundamentals

June 1980

This manual teaches the fundamentals of computer programming, using MINC and the BASIC programming language. *Book 3: MINC Programming Reference* should be used in conjunction with this book for technical details and quick reference.

This manual supersedes *Book 2: MINC Programming Fundamentals*, Order Number AA-D799A-TC.

Order Number AA-D799B-TC

MINC–11

VERSION 1.2

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

NORTHEAST/MID-ATLANTIC REGION

Technical Documentation Center
Cotton Road
Nashua, NH 03060
Telephone: (800) 258-1710
New Hampshire residents: (603) 884-6660

CENTRAL REGION

Technical Documentation Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone: (312) 640-5612

WESTERN REGION

Technical Documentation Center
2525 Augustine Drive
Santa Clara, California 95051
Telephone: (408) 984-0200

digital equipment corporation • marlboro, massachusetts

First Printing, November 1978
Revised, June 1980

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsbility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DECnet	IAS
DECUS	DECsystem–10	MASSBUS
Digital Logo	DECSYSTEM–20	PDT
PDP	DECwriter	RSTS
UNIBUS	DIBOL	RSX
VAX	EduSystem	VMS
MINC–11	VT	

CONTENTS

FIGURES

CHAPTER 1
INTRODUCTION

This book teaches you the commands necessary to use your MINC system. However, this book teaches more than just the syntax of the commands — it also teaches you how to put these commands together to produce your desired results. This book teaches you good techniques for making MINC work more effectively as well as for putting the commands together more effectively.

This book is designed to be read sequentially from the front to the back because the later chapters build on your knowledge from the earlier ones. Consequently, the later chapters are not all easy to understand. In order to show you how to use your MINC effectively, some of the later examples become somewhat complex. However, the examples are explained quite thoroughly; and if you take the time to understand the examples, you will finish this book with a good knowledge of your MINC and how to use it.

All of the examples in this book have been chosen carefully. Examples can never reflect every possible environment, but the ones presented in this manual all reflect possible uses for MINC in the laboratory environment. Many of the examples are actually tools that you will want to use in your own work with MINC.

This book discusses all of the MINC commands but does not discuss all of the technical details of each command. The more complex and technical points of the commands are left for *Book 3: MINC Programming Reference.*

This book repeatedly makes reference to Book 3. Feel free to look up a subject in Book 3 for further information. Book 3 is arranged in alphabetical order by topic for easy reference.

WHAT IS MINC?

MINC, which stands for Modular INstrument Computer, is both a calculator and a computer. When MINC is ready to accept commands from you, it displays a READY message. MINC has two modes: the *immediate mode* and the *program mode*. In the immediate mode, MINC performs the instructions immediately as you type them. In the program mode, MINC saves the instructions that you type and performs them only when you request it.

You communicate with MINC through the terminal. You *input* information to MINC by typing on the terminal keyboard, and MINC *outputs* information to you by displaying its messages on the terminal screen.

Notice the flashing box on the terminal screen. This flashing box is called the cursor. If anything happens on the screen, it happens where the cursor is. When you type a character on the screen, MINC displays the character at the current cursor location and then moves the cursor to the right, one place.

You can type a command to MINC whenever MINC is displaying the READY message. You terminate a command by pressing the RETURN key. If you are in the immediate mode when you press the RETURN key, MINC immediately tries to *execute* the command. If you are in the program mode when you press the RETURN key, MINC does not try to execute the command until you tell it to.

You cannot damage MINC by typing an incorrect command. If you notice a mistake before you press RETURN, you can erase your mistake by pressing the DELETE key until the mistake is erased, and then you can type in the correct command.

If you press RETURN before you notice the mistake, MINC will tell you that you made a mistake. Then MINC will display the READY message and will wait for you to type the command again.

Using the Terminal

You type commands to MINC using the terminal keyboard, which is like a typewriter. MINC understands commands in both upper and lower case.

When you type a character, MINC reads it and then displays it on the terminal screen. Usually, when you type a lower case character, MINC displays a lower case character. Sometimes, however, when MINC displays a question for you to answer, it will display your response in upper case, even when you type your response in lower case characters.

The BASIC Language

You communicate with MINC using the BASIC language. Basic is a language that is like English but has a limited vocabulary. In BASIC, each word in the vocabulary has only one meaning to MINC.

For example, in BASIC, the verb PRINT tells MINC to display something on the terminal screen. You tell MINC what to print, and MINC prints it.

On the other hand, in English, the verb print can mean one of the following:

- Produce a page of text using a printing press (like to print a newspaper).
- To write in a certain manner — a child is taught to print before being taught to write in script.
- To produce a photograph on paper from a negative.

Thus, BASIC is less ambiguous than English and is easier for MINC to understand.

Due to the limited vocabulary, you must be careful to type exactly what you mean. For example, you cannot use a lower case ell (l) instead of a one (1). A lower case ell (l) means something very different to MINC than a one (1) does. Similarly, you cannot use a capital O (oh) in place of a 0 (zero).

Commands and Statements

The BASIC language gives two types of instructions to MINC: *commands* and *statements*. Until now, the term command has been used loosely to mean any instruction given to MINC. However, there is a difference between commands and statements. A BASIC *command* can be used only in the immediate mode. A BASIC *statement* can be used in either the immediate mode or the program mode.

You can put a series of BASIC statements together to form a

program. Thus, a program is just a group of statements that perform a larger task than you can perform with just one statement.

The MINC Workspace

MINC has a *workspace* in which it stores a program that you type and any values that you tell it to save. The workspace has a limited size, and thus, can only hold a limited amount of information.

MINC does not save the results of any command in the workspace, but it does save the results of some statements — even in the immediate mode. As you type in a program, MINC saves that program in the workspace. However, MINC can only save one program in the workspace at a time. If you type in a new program, you destroy the old program in the workspace.

Special Terminal Keys

There are some terminal keys that have special meanings to MINC.

The RETURN Key The RETURN key signals the end of your command or statement. In examples, this key is represented by the following symbol:

(RET)

This symbol appears only in the first few examples in this manual to show you where to use it. In later examples, you should remember to press the RETURN key at the end of the line shown in the example.

The NO SCROLL Key The NO SCROLL stops all output from appearing on the terminal screen. If you press the NO SCROLL key a second time, the output resumes on the screen. Thus, if you are typing on the terminal and nothing appears on the screen, press the NO SCROLL key, just in case you pressed the NO SCROLL key by mistake. The NO SCROLL key is an on-off switch for the terminal output.

Control Characters There are special characters called *control characters*. For example, you send MINC a "Control C" by pressing the CTRL key at the same time as the C key (similar to sending a capital C by pressing the SHIFT key at the same time as the C key). In a paragraph the "Control C" character is represented as CTRL/C. In an example, it is shown as:

(CTRL/C)

When you press CTRL/C, the following symbol appears on the screen.

^C

The control characters that are important to MINC are: CTRL/C, CTRL/S, CTRL/Q, and CTRL/U. There are other control characters that have significance, but the other control characters are explained in Book 3.

The CTRL/C character stops a program. This character is explained in Chapter 3.

The CTRL/S character stops all output from appearing on the terminal screen. The CTRL/Q character shows all the output that was hidden with the CTRL/S character. Thus, if you are typing on the terminal and nothing appears on the screen, press CTRL/Q, just in case you typed CTRL/S by mistake.

The CTRL/S and CTRL/Q characters together are the same as the NO SCROLL key.

The CTRL/U key signals MINC to ignore the current line that you are typing. When you press CTRL/U, MINC returns the cursor to the next line. The line you want to erase still appears on the screen, but MINC ignores it.

MINC VOLUMES

Before you can use your MINC, you must place a MINC system diskette in the left diskette drive. The diskette is a more permanent storage medium than the MINC workspace. If you do not have a MINC system diskette, read *Book 1: Introduction to MINC* to learn how to create a system diskette from the Master diskette.

The diskette is a physical medium for storage. A *volume* is a logical storage medium to MINC. For example, a volume to a person can be one book in a set of encyclopedias — any one of the set, not one in particular. A volume to MINC is any storage medium — not just one particular diskette. Thus, the term volume is used throughout this manual to refer to a diskette.

MINC has logical devices as well as logical storage media. The diskette drive is a physical device. The diskette is placed in a diskette drive. The logical names for the two diskette drives are SY0: and SY1: (the colon is part of the name). SY0: is the name

of the left diskette drive, and SY1: is the name of the right diskette drive.

USING THIS MANUAL

To learn BASIC most effectively, you should read this book and try the examples on your MINC as you read them. The best way to learn BASIC is to use it.

To try the examples in this book, you must have a demonstration diskette in the left diskette drive (SY0:). The directions for starting your MINC system and obtaining a copy of the demonstration diskette are in Part II of *Book 1: Introduction to MINC*.

CHAPTER 2
USING MINC IN THE IMMEDIATE MODE

The immediate mode is used for two purposes:

- To display calculations.
- To command MINC to perform tasks such as running and saving programs.

This chapter describes how to use MINC to do calculations. Chapter 4 describes the immediate mode commands that direct MINC to perform other tasks.

Imagine yourself using a small calculator to do the following calculation:

$$\frac{(27+32)(15-8)}{327}$$

you would probably follow these steps:

1. Enter 27.
2. Add 32.
3. Store result (in memory or on paper)
4. Enter 15.
5. Subtract 8.
6. Multiply by what is stored in memory or on paper.
7. Divide by 327.

At this point the calculator display is showing the correct answer, which is 1.263.

```
PRINT 23.8
  23.8

PRINT 5324.7+78625.9
  83950.6
```

You can insert blanks or spaces to make the PRINT statement more readable. MINC ignores blanks. All of the following examples have the same meaning to MINC.

```
PRINT 7

PRINT7

P R I N T 7
```

You can also type the PRINT statement in any mixture of upper and lower case. All of the following examples have the same meaning to MINC.

```
prINT 7

print 7

PRINT 7
```

However, in this manual, all MINC statements are typed in upper case to distinquish them from other parts of the text.

Numeric Literals

A single number can be the expression of a PRINT statement. Another name for a number is a *numeric literal.* Any number whose magnitude is between .01 and 999999 can be printed just the way it looks. Zero is also printed the way it looks. For example:

```
PRINT .01
  .01

PRINT 999999
  999999

PRINT -.01
-.01

PRINT 0
  0
```

Numeric literals cannot contain commas or spaces. Numeric

literals can contain up to six significant digits; that is, numeric literals can have up to six digits, not counting zeroes preceding the number or zeroes trailing the decimal point. Some valid numeric literals are:

0.123456

999999

99999.9

123.456

In a PRINT statement, MINC rounds decimal places to six significant digits. That is, if the value of the seventh digit (last decimal place) is five or less, the remaining digits are lost; if the value of the seventh digit is six or greater, the sixth digit is incremented by one. For example:

PRINT Statement	*What Is Printed*
PRINT .1234565	.123456
PRINT .1234566	.123457
PRINT 1.2345672	1.23457

Numbers that have a magnitude greater than 999999 or less than .01 are printed in scientific notation, where E denotes the exponent, or power of 10, involved. For example:

PRINT Statement	*What Is Printed*
PRINT .01	.01
PRINT .0099	9.90000E-03 (9.9×10^{-3})
PRINT 999999	999999
PRINT 1000000	1.00000E+06 (1×10^{6})
PRINT .0100	.01
PRINT 090	90
PRINT 1E-01	.1
PRINT 1E6	1.00000E+06 (1×10^{6})

Examples 5 and 6 show that MINC removes preceding and trailing zeroes. Examples 7 and 8 show that you can used E notation in a PRINT statement. In Example 7, MINC converts the E notation to .1 because the value is between .01 and 999999. In Example 8, MINC converts the result to 1.00000E+06 because this is the form in which MINC prints E notation, even though MINC understands shortcuts.

The range of magnitudes that can be represented in E notation

is between 10^{-38} and 10^{38}.

Arithmetic Operations and Priority of Operators

MINC performs the following arithmetic operations: addition, subtraction, multiplication, division, and exponentiation. In order to obtain the correct result of a calculation, you should know the order in which MINC performs these operations. MINC performs calculations in a certain order unless forced by you to do otherwise. All MINC calculations must be written on one line.

Any of the arithmetic operators (described in the following sections) combined with numeric literals are valid expressions. Expressions containing arithmetic operations are called *arithmetic expressions*.

Addition and Subtraction Addition is denoted by a plus sign (+). Sums are obtained by putting those numbers to be added in a PRINT statement with plus signs between them. For example:

```
PRINT 7+2
  9

PRINT 4.3+9.2+8.7
  22.2

PRINT 6E6 + 5E5
  6.50000E+06
```

Subtraction is denoted by a minus sign (-). Differences are obtained by putting those numbers to be subtracted in a PRINT statement with minus signs between them. For example:

```
PRINT 8-5
  3

PRINT 9.4-6.5-2
  .9

PRINT 5.3E-7 - 4.9E6
-4.90000E+06
```

Notice that in the last example, 0.00000053 is negligible next to 4,900,000.

Addition and subtraction are of equal priority; that is, if MINC is given a series of additions and subtractions, it proceeds from left to right. For example:

```
PRINT 5+7-2
  10
```

```
PRINT 6-2-3+4
  5

PRINT 4.9-3.2+6.7-10.9
-2.5
```

In the second example, the first subtraction (6-2) is done first, then the second subtraction (4-3), and finally the addition (1+4).

Multiplication and Division Multiplication is denoted by an asterisk (*) rather than an ×, and the * must never be left out if a multiplication is to be performed. In mathematical texts, multiplication is represented in several ways; sometimes with an ×, sometimes with a dot, and sometimes by writing expressions without any operator.

In MINC, the only way to obtain the product of two numbers is to insert an asterisk (*) between the numbers to be multiplied. For example:

```
2*3
```

You can obtain products of numbers by putting those numbers to be multiplied in a PRINT statement. For example:

```
PRINT 8*5
  40

PRINT 4.35*6.72
  29.232

PRINT 6E4*3E9
  1.80000E+14
```

The multiplication operation has a higher priority than addition and subtraction and thus is done first when combined with addition or subtraction. For example:

```
PRINT 2+3*7
  23
```

MINC does the multiplication (3*7) first and then the addition (2+21). Another example:

```
PRINT 2+3*7-9
  14
```


In this case, MINC does the multiplication (3*7) first, and then

the addition (2 + 21) and subtraction (23-9) are done from left to right since they are of equal priority.

Division is denoted by a slash (/) and the divisor must appear on the same line as the dividend. For example, three-sevenths in mathematical notation must appear as 3/7 in MINC notation.

Examples of division are:

```
PRINT 3/5
  .6

PRINT 10/3
  3.33333

PRINT 20/3
  6.66667

PRINT 10E6/5E4/2
  100

PRINT 6/0
?MINC-W-Dividing by zero
  0
```

In the final example, an error message notes that division by zero is not defined and consequently cannot be computed.

Multiplication and division are operations of equal priority. When they are combined, MINC performs these operations from left to right. For example:

```
PRINT 2*3/9
  .666667

PRINT 4/10*5
  2

PRINT 5*4/10
  2

PRINT 5E8 * 2E7 / 9E6
  1.11111E+09
```

Division, like multiplication, is of higher priority than addition and subtraction, and MINC performs division before addition or subtraction. For example:

```
PRINT 5/2+3
  5.5
```

```
PRINT 2+3*5-6/3
  15
```

In the second example, the multiplication (3*5) is done first, then the division (6/3), and then the addition and subtraction proceed from left to right.

Exponentiation In mathematics, to raise 2 to the third power, you write:

2^3

Because calculations in MINC must be written on one line, exponentiation is denoted by a caret (^), and thus to raise 2 to the third power you write:

2^3

```
PRINT 2^3
  8
```

MINC processes exponentiation operations, like the other operations, from left to right. For example:

```
PRINT 2^3^2
  64
```

First MINC computes 2^3 (=8) and then squares the result (=64).

Overriding Priorities You can override the priority of multiplication and division over addition and subtraction by the use of parentheses. Anything in parentheses has priority over anything outside parentheses. In the following two examples, see how the parentheses affect the result.

```
PRINT 2+3*7
  23

PRINT (2+3)*7
  35
```

In the second example, MINC performs the addition (2+3) first because this operation is enclosed in parentheses. Another example:

```
PRINT 6+(2+7)/(2+1)-4
  5
```

MINC performs the addition (2+7) first, then the addition (2+1), because they are in parentheses; then MINC performs the division because it has the next higher priority; and finally MINC performs the addition (6+3) and subtraction (9-4) from left to right.

You can place parentheses within parentheses, to *nest* one expression within another. For example:

```
PRINT (2+7*(6+5))/24+5
  8.29167
```

MINC performs the innermost addition (6+5) first. Then MINC performs the operations within the outer parentheses according to the usual priorities. Finally, MINC performs the operations outside the parentheses according to the usual priorities. That is:

(2+7*(6+5))/24+5

6+5=11

7*11=77

2+77=79

79/24=3.29167

3.29167+5=8.29167 The Answer

Note that every left parenthesis must be matched somewhere in the expression by a right parenthesis.

Table 1. Priority of Operators

Operation	*Priority*
( )	highest
^	
* or /	
+ or -	lowest

Exponentiation has the highest priority of all the operations as shown in Table 1, but like the rest, that priority can be overridden by parentheses. For example:

```
PRINT 2+3^3
  29
```

```
PRINT (2+3)^3
  125

PRINT (2^3)^2
  64

PRINT 2^(3^2)
  512

PRINT 2*2^3
  16

PRINT (2*2)^3
  64
```

SOME MINC FUNCTIONS

MINC can perform some functions that are difficult to calculate by hand. For example, MINC can compute square roots, logarithms, and trigonometric functions.

A *function* is an operation on an argument or arguments that computes a value. For example, the symbol

$\sqrt{x}$

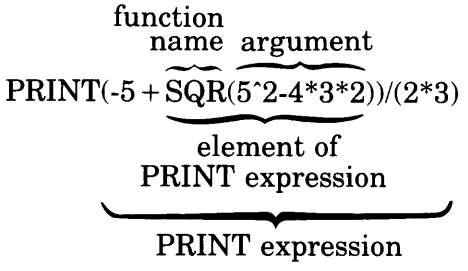
represents the value equal to the square root of x. The x is the *argument* of the square root function.

To print the value of the square root of 2 on MINC, type:

```
PRINT SQR(2)
  1.41421
```

MINC prints the value of the square root of 2 on the terminal screen.

A function, which consists of the function name followed by a parenthesized expression called an *argument*, can be used as an expression of a PRINT statement, as above. A function can also be used with a larger expression wherever an expression is valid. For example:



MINC functions calculate results to six significant digits.

Some MINC functions are described in the following sections. The remaining functions are described in Chapter 8.

The Square Root Function

The square root function, denoted as SQR, calculates the value of the square root of the argument. The following four examples show uses of the SQR function. The third example shows that the square root function can be used as the argument of the square root function; that is, function calls can be *nested.* The fourth example shows that an expression can be the argument of the SQR function as well as the SQR function being an element of the larger expression of the PRINT statement.

```
PRINT SQR(4)
  2

PRINT SQR(2)
  1.41421

PRINT SQR(SQR(16))
  2

PRINT (-5+SQR(5^2-4*3*2))/(2*3)
-.666667
```

As with all MINC operations, the precision of the result of the square root function is to six significant digits.

Trigonometric Functions

The PI function always takes on the value of π to six significant digits (3.14159). The PI function has no argument and can be used wherever the value 3.14159 is required.

The sine, cosine, and arc tangent functions, which are denoted by SIN, COS, and ATN respectively, calculate the value of the sine, cosine, and arc tangent of their arguments respectively.

The arguments of both the sine and cosine functions are angles that must be expressed in radians. You can convert an angle in degrees to radians with the following identity:

$$\text{radians} = \text{degrees} * PI/180$$

where PI is the function that takes on the value 3.14159.

The form of the sine function is:

SIN(expression)

The form of the cosine function is:

COS(expression)

For example:

```
PRINT SIN(PI)
  1.87254E-07

PRINT COS(PI)
-1
```

Notice that MINC returns a value of .0000001 for the sine of π. This value is approximately 0. See the Numeric Precision section of Book 3.

If you want the value of the sine of 239 degrees, type:

```
PRINT SIN(239*PI/180)
-.857167
```

As with the SQR function, you can nest the trigonometric functions. For example,

```
PRINT SQR(COS(45*PI/180))
  .840896
```

The form of the arc tangent function is:

ATN(expression)

The ATN function calculates the value in radians in the range $+\pi/2$ to $-\pi/2$. For example:

```
PRINT ATN(32.435)
  1.53998
```

You must compute the other trigonometric functions — tangent, cotangent, arc sine, arc cosine — using the SIN, COS, and ATN functions. For example, the following PRINT statement prints the tangent of $\pi/3$.

```
PRINT SIN(PI/3)/COS(PI/3)
  1.73205
```

Exponential Function

The exponential function raises the number *e* (approximately 2.71828) to the power specified by the argument of the function. That is, EXP(7) is equivalent to e^7.

The form of the exponential function is:

EXP(expression)

For example:

```
PRINT EXP(7)
  1096.63

PRINT EXP(SQR(2+3))
  9.35647
```

Logarithmic Functions

The LOG function calculates the value of

$\log_e$(expression)

The LOG function is the inverse of the EXP function since the following relationship is true:

LOG(EXP(expression)) = expression

For example:

```
PRINT LOG(2.718281)
  1

PRINT LOG(EXP(5))
  5
```

The LOG10 function returns the value of

$\log_{10}$(expression)

For example:

```
PRINT LOG10(10)
  1

PRINT LOG10(100)
  2
```

Logarithms to any base may be easily computed using the following formula:

$$\log_a(\text{expression}) = \frac{\log_e(\text{expression})}{\log_e(a)}$$

For example, the MINC statement to compute the log base 2 of 512 is:

```
PRINT LOG (512)/LOG(2)
  9
```

VARIABLES

Up to now, all of the calculations have involved numeric literals. However, most mathematical formulas have variables in them — that is, quantities that can vary. For example, the following formula represents the area of a circle.

$$\text{area} = \pi r^2$$

The π represents the number 3.14159 and the *r* represents the radius of the circle. The *r* is a variable — that is, it is a placeholder whose value varies for different circles. On the other hand, π is a constant whose value is always 3.14159.

Similarly, the formula for conversion from Fahrenheit to Celsius has constants and variables.

$$T_c = 5/9*(T_f-32)$$

T_c is the variable representing the result in degrees Celsius and T_f represents degrees Fahrenheit. The numbers 32 and 5/9 are numeric literals.

In MINC, a *numeric variable* is the name of storage for a numeric value. The storage is allocated in the workspace.

You can use variables in the immediate mode to save space on a line in a long calculation and to cut down on typographical errors. If you are going to do several calculations with the same large expression, you can save time by storing the value of the expression in a variable.

For example, if you want the sine, cosine, tangent, and arc tangent of 45 degrees, you might type the following:

```
PRINT SIN(45*PI/180)
PRINT COS(45*PI/180)
PRINT SIN(45*PI/180)/COS(45*PI/180)
PRINT ATN(45*PI/180)
```

However, using the variable A to represent 45 degrees converted to radians, you need only type:

```
A=45*PI/180
PRINT SIN(A)
PRINT COS(A)
PRINT SIN(A)/COS(A)
PRINT ATN(A)
```

MINC represents numeric variables with a letter or a letter followed by a digit. For example, the variable names F and F1 represent two distinct variables. You can print the area of a circle with radius 5 by using the following statements:

```
R=5

PRINT PI*R^2
```

In MINC, a numeric variable represents a numeric value. In the previous example, the variable R has a numeric value of 5. The next section describes how to assign a value to a variable.

Assigning Values to Variables

You can assign a value to a variable with an *assignment statement*. The form of the assignment statement is:

LET variable-name = expression

The keyword LET is optional. The value on the left of the equals sign must be a variable name. The value on the right side of the equals sign must be a valid expression. There is another type of numeric variable called an *integer variable* which can contain only whole number values. Integer variables are discussed in Chapter 8.

The following example shows assigning the value 7 to the variable R.

```
R=7
```

Hereafter, whenever you use the variable R, MINC goes to the workspace and uses the value 7. For example:

```
PRINT PI*R^2
  153.938
```

The equals sign (=) does not denote mathematical equality. Instead, in MINC it means, "take the value to the right of the

equals sign and place it in the variable to the left of the equals sign." For example, both of the following statements are acceptable to MINC, but the second is not valid in mathematics.

```
A=2

A=A+1
```

The first statement tells MINC to give variable A the value 2. The second statement tells MINC to calculate the value of A + 1 (which is 2 + 1 or 3), and then to place this value in the variable A. At the end of statement 2, A has the value 3.

VARIABLES AND THE MINC WORKSPACE

As stated earlier, MINC stores variables in the workspace. Whenever you use a variable, MINC sets up a place in the workspace for it. The value of the variable is zero until you assign a new value to it.

When you turn MINC off, all the variables you used are erased from the workspace. Thus, when you turn MINC on, there are no variables until you create one by assigning a value to it.

The CLEAR Command

To reset all variables to zero, use the CLEAR command. The form of the CLEAR command is:

CLEAR

When you use the CLEAR command, your variables still exist in the workspace, but MINC has set their values to zero.

The Scratch Command

The SCR command, which stands for scratch, erases the entire workspace. The form of the SCR command is:

SCR

When you use the SCR command, MINC zeroes all variables and erases any program you stored in the workspace. (See Chapter 3.)

MULTIPLE STATEMENT LINES

You can type more than one BASIC statement on a line on the screen by using the backslash (\). The form of the backslash is:

statement \ statement

The backslash works only for BASIC statements and not for commands. Note that the backslash (\) is different from the slash (/), which indicates division.

For example:

```
A=45*PI/180 \ PRINT SIN(A) \ PRINT COS(A) \ PRINT ATN(A)

.707107
.707107
.665774

READY
```

By placing more than one statement on a line, you can see all of the results together instead of separated by blank lines and READYs.

Note that you can only put as many statements on a line as will fit. Remember that BASIC statements cannot cross line boundaries.

CHAPTER 3
USING MINC IN THE PROGRAM MODE

The advantage of a computer is its ability to perform repetitive or iterative tasks automatically. As a simple example, consider computing the sine and cosine of many angles in degrees. When using MINC in the immediate mode to compute the sines and cosines of 45°, 272°, 130°, and 90°, you would have to type all of the following lines. The comma between expressions instructs MINC to print two numbers on the same line.

```
A=PI/180

PRINT SIN(45*A),COS(45*A)
  .707107      .707107

PRINT SIN(272*A),COS(272*A)
  -.999391      .034892

PRINT SIN(130*A),COS(130*A)
  .766045      -.642788

PRINT SIN(90*A),COS(90*A)
  1      0
```

This is a lot of typing, and if you need the sines and cosines of more angles, you will have to do more typing and will have more chance for error. You can type a BASIC program that will compute the sine and cosine of any angle in degrees, but the program needs to be typed only once.

A *program* is a series of statements that can be executed as a whole. It can perform more than one calculation or task.

A PROGRAM EXAMPLE

Suppose a physics student has a MINC available to her for her homework. Since she needs to compute sines and cosines for many of her physics problems, she wrote a program to compute as many sines and cosines as she needs. Her program asks for an angle in degrees and then prints out the sine and cosine. The program keeps asking for another angle until she types in a CTRL/C (presses the CTRL key at the same time as the C key).

The student's program follows. You should type it in just to see how a program works. *Do not worry if you do not quite understand the program.* All of the programming concepts shown here are explained later in this manual.

What you type in is shown in red ink. MINC's responses are shown in black.

This program should be typed exactly as it appears here. You must type each statement perfectly. If you make a typographical error, simply retype the line. MINC executes the statements in numerical order, even if you did not enter them in order. Remember that spaces within a statement are there only to make the statement easier to read — they do not matter to MINC.

```
READY
NEW
NEW FILE NAME -- SINES
```

The NEW command erases and then names the workspace. Thus, if you want to save the program you are typing in, give the program a name. In this case the workspace is named SINES.

```
READY

20 PRINT ,"SINE","COSINE"
30 PRINT "DEGREES";
40 INPUT X
60 Z=X*PI/180
70 PRINT ,SIN(Z),COS(Z)
80 GO TO 30
90 END
```

If your program is not typed perfectly, retype lines that are not correct by retyping the line number followed by the statement. For example, if you had typed:

```
40 INPUT C
```

you can now type:

```
40 INPUT X
```

and MINC will correct the line by replacing the old line 40 with the new one.

To see the way your correct program looks, type:

```
LIST
```

The LIST command causes your entire program to appear on your terminal screen.

When you are convinced that your program is typed properly, type:

```
RUN
```

MINC will then *run* or *execute* your program; in other words, MINC will perform the instructions you have given it.

Output from the above program follows. What MINC types on the screen is represented in black ink while your responses are represented in red. After entering each response, press the RETURN key.

```
SINES     20-APR-80          03:28:32

                             SINE              COSINE
DEGREES? 45                  .707107           .707107
DEGREES? 272                 -.999391          .0348992
DEGREES? 130                 .766045           -.642788
DEGREES? 90                  1                 0
DEGREES? ^C
STOP at line 40
READY
```

If you type RUN, MINC will perform this program again; you can enter more angles and MINC will compute more sines and cosines. You can also save this program and call it back when you need more sines and cosines, saving yourself many steps. (How to save programs is discussed in Chapter 4.)

PROGRAMS AND THE MINC WORKSPACE

You can run the program many times because MINC stores programs in the workspace as you type them. Before you type in a new program, you should make sure that any previous program is erased from the workspace by typing the SCR command (explained in Chapter 2) or the NEW command (explained in the following section).

When you type a program, MINC stores the program in the workspace in the upper case, whether you typed in the program in upper case or lower case. MINC also formats the program for you. MINC leaves at most one space between words, no matter how you type in the program.

The NEW Command

Like the SCR command, the NEW command erases the workspace. However, the NEW command also allows you to give the workspace a name. The SCR command automatically names the workspace NONAME.

The NEW command has two forms. If you type only NEW followed by RETURN, MINC prompts you for the new workspace name.

```
NEW(RET)
NEW FILE NAME - -
```

Here you type in a name from one to six characters long. The name can be any combination of letters and digits.

The second form of the NEW command is:

NEW name

where name is the name that you want (from one to six characters). In this second case, MINC does not prompt you for the name.

You should use the NEW or the SCR command to erase the SINES program from the workspace before you try the next examples. Unlike the NEW command, the SCR command automatically names the workspace NONAME. Thus, if you care about the name of the workspace, use the NEW command.

STATEMENT NUMBERS

A statement number preceding a BASIC statement determines whether MINC is in the immediate mode or the program mode. For example:

```
PRINT 7
```

causes MINC to print a 7 immediately, while:

```
10 PRINT 7
```

does not cause MINC to do anything immediately. When you type in a statement with a statement number, MINC does not

execute the statement, but rather places the statement in the workspace. The above example is a one-line program consisting of one PRINT statement. To execute a program (for example, 10 PRINT 7), you must type the RUN command. For example:

```
READY
SCR

READY
10 PRINT 7
RUN

NONAME              20-APR-80          04:46:28

 7

READY
```

The program mode is much more powerful than the immediate mode because it permits MINC to perform a series of statements together as in the example in the beginning of the chapter.

Statement numbers tell MINC the order in which to execute the BASIC statements. MINC executes the statements in ascending numerical order, whether you entered the statements in numerical order or not. The numbers do not have to be consecutive. In fact, it is wise to leave a gap between statement numbers in case you omitted a statement and have to insert it later.

INPUT STATEMENTS

The main purpose of a computer is to accept data as input, process the data, and produce output that is readable and usable.

Assignment statements assign values to variables, these values being determined by the program. In contrast, INPUT statements stop a program to accept data values from the user (the person using the program — possibly someone other than the programmer).

If you use variables along with the INPUT statement a program can process any value that you input, as long as it is within the boundaries of the program. The following example demonstrates how to use the INPUT statement.

Suppose an engineer needs to find the volume of many cylinders. For each cylinder, the engineer must know the area of the base and the volume. The area of a circle is:

$$\pi r^2$$

where r is the radius of the circle (base). The volume of a cylinder is:

$$\pi r^2 h$$

where h is the height of the cylinder.

To find the area and volume of a cylinder with radius 2cm and height 5cm, the engineer could enter the following line into MINC.

```
PRINT PI*2^2, PI*2^2*5
```

For each cylinder having different dimensions, the engineer would have to enter this same line with the new dimensions. If, however, the engineer wrote a program to calculate the area and volume of the cylinder, he would have to type the program only once and it would take as input the radius and height and would print the area of the cylinder base and volume of each cylinder. The following program would do this:

```
NEW CYLIND

READY

10 INPUT R,H
20 PRINT PI*R^2, PI*R^2*H
```

When the engineer types RUN, the program, from statement 10, will produce a question mark as a prompt indicating that MINC is waiting for him to type in the radius and the height separated by commas. The engineer must press the RETURN key to signify the end of his input line. Then, underneath these input amounts, MINC will print the area and volume of the cylinder. The screen would look like this:

```
RUN

CYLIND                  20-APR-80               04:48:37

?2,5

12.5664         62.8319
```

Red represents what the engineer typed and black represents what MINC typed.

In this example, the INPUT statement assigns to the variables

R and H the values that the engineer entered at the question mark. Now, in line 20, MINC uses 2 and 5 as the values for the variables R and H respectively, to compute the area of 12.5664 and volume of 62.8319.

The form of the INPUT statement is:

INPUT variable-list

where variable-list is one variable or, as in this example, many variables separated by commas. For example:

```
10 INPUT F
20 INPUT A,B,C1
```

In statement 10, MINC gives a question mark as a prompt for you to enter one piece of information followed by a RETURN. An INPUT statement produces one question mark for each variable it expects. In statement 20, MINC issues a question mark and expects three numbers separated by commas or RETURNs. In the second case, if you have entered only two numbers before you pressed RETURN, MINC issues another question mark for the third number.

When you type RUN and MINC starts to execute the program, all variables have the value zero. If there is an INPUT statement in the program, then those variables listed in the INPUT statement are given the new values that you enter at the question mark.

ARITHMETIC OPERATIONS AND PRIORITY OF OPERATORS

The arithmetic operations, addition, subtraction, multiplication, division, and exponentiation, are the same in the program mode as in the immediate mode; the order of priority being: exponentiation as the highest, multiplication and division second, and addition and subtraction lowest.

At the time MINC performs a calculation, it treats a variable as a number, giving it the value you entered in the INPUT statement. The priority of the operations remains the same with variables as with numbers.

MAKING OUTPUT READABLE

The engineer in the earlier example decided that he was not satisfied with his program for calculating the area and volume of his cylinders. He did not like the fact that the only prompt for input was a question mark which required him to remember

what he had to type in. A much more meaningful dialog on the terminal would be:

```
CYLIND                21-APR-80              09:43:03

Radius? 2
Height? 5
Area: 12.5664
Volume: 62.8319
```

In this case, he would know easily what he should type in, and the results are clearly labelled. The changes to his program are minimal. His original program was:

```
10 INPUT R,H
20 PRINT PI*R^2, PI*R^2*H
```

His new version of the program is:

```
CYLIND                21-APR-80              10:27:27

10 PRINT 'Radius'; \ INPUT R
20 PRINT 'Height'; \ INPUT H
30 PRINT 'Area:'; PI*R^2
40 PRINT 'Volume:'; PI*R^2*H

READY
```

and the output is the same as shown previously.

These new changes are explained in the next two sections.

String Literals

Statement 10 of the engineer's new program prints out the label "Radius". This label is enclosed in apostrophes within the PRINT statement and is called a *string literal.* A string literal can be alphabetic letters or numbers or both.

Digits in a string literal are characters rather than numbers. Notice the difference in the following two PRINT statements. The first prints a numeric value, so MINC computes the value and prints out the result, which is 5. The second prints a string literal. PRINT "2+3" means to print the character "2" followed by the character "+" followed by the character "3".

```
PRINT 2+3
  5

PRINT '2+3'
  2+3
```

When you are entering string literals, you must enclose them in a pair of apostrophes (') or in a pair of quotation marks ("). If you need to use an apostrophe within a string literal, then you should use quotation marks to delimit the string literal. If you need to use quotation marks within a string literal, then you should use apostrophes to delimit the string literal. In all other cases, you can use either apostrophes or quotation marks, but you cannot use an apostrophe at one end of the string literal and a quotation mark at the other. For example.

```
PRINT "THE BOY'S COAT"
THE BOY'S COAT
```

Although spaces have no meaning within BASIC statements, within string literals spaces are significant. In the second of the two following examples, the space character is part of the string constant because it is enclosed within the quotation marks.

```
P  R  I  N  T  2  +  3
  5

PRINT "2  +  3"
  2  +  3
```

In the following example, notice that MINC does not capitalize lower case letters within string literals.

Examples:

```
10 PRINT "the area"
20 PRINT 'of a circle is'
30 PRINT "computed using the"
40 PRINT 'formula pi r squared'
RUN

NONAME                  12-JUN-80               11:20:15

the area
of a circle is
computed using the
formula pi r squared
```

Print Zones

In line 10 of the program CYLIND, the engineer placed a semicolon (;) after his string literal. The semicolon prevents the terminal from going to a new line before printing the question mark prompt from the INPUT statement. In the second of the

two partial examples below, there is no semicolon after RADIUS and consequently the prompt from the INPUT statement is printed on the next line.

```
10 PRINT 'Radius';
20 INPUT R
RUN

NONAME              24-APR-80              10:30:46

Radius?

READY

10 PRINT 'Radius'
20 INPUT R
RUN

NONAME              24-APR-80              10:31:15

Radius
?
```

MINC considers the terminal screen to be divided into five 14-space columns or zones. A comma in a PRINT statement causes the terminal to skip to the next print zone. For instance, in the following example, the first comma causes the terminal to skip to the second zone, SINE is printed in the second zone, the second comma causes the terminal to skip to the third zone, and COSINE is printed in the third zone.

```
PRINT ,'SINE','COSINE'
              SINE          COSINE
```

Nothing is printed in the first zone (because the initial comma causes the terminal to skip the first zone).

Two successive commas skip an entire zone. For example, the following PRINT statement would print A in the first zone and B in the third zone.

```
PRINT 'A',,'B'
A                B
```

In the next example, MINC prints 6 on the next line because there are only five print zones per line.

```
PRINT 1,2,3,4,5,6
 1            2            3            4            5
 6
```

To suppress the print zones, use the semicolon. For example:

```
PRINT 1;2;3,4,5,6
 1  2  3             4          5          6
```

Because there is a semicolon between 1 and 2 and between 2 and 3, MINC prints the 1, 2, and 3 in the same zone. MINC leaves one space before each number for a sign (in this case space for positive) and one space after each number as a separator.

The comma between 3 and 4 causes the 4 to be printed in zone two. The 5 and 6 are printed in zones three and four because of the commas.

Note that a semicolon puts no spaces between string constants. For example:

```
PRINT 'A';'B'
AB
```

To skip lines, use PRINT statements without arguments — one PRINT statement for each skipped line, as shown below.

```
PRINT
```

If you end the PRINT statement with a semicolon, MINC does not start the next PRINT statement on a new line. For example:

```
10 PRINT 1;
20 PRINT 2;
30 PRINT 3;
40 PRINT 'this is the end of the line'

READY
RUN

NONAME              20-JUL-80           11:40:05

1 2 3 this is the end of the line

READY
```

The TAB Function

The TAB function causes the terminal to skip to a specified column within the line to be printed. The form of the TAB function is:

PRINT TAB(expression);

where expression is a column number greater than 0. If the expression is greater than the number of columns on a line, MINC continues to skip lines until it can tab to a column within the line. If the column number specified in the expression is less than 0, the following error message is printed.

```
?MINC-F-Arguments in definition do not match function called
```

If the expression is not an integer, MINC uses only the integer portion of the number. In the following two examples, a hyphen (–) represents a space, so that you can count the spaces more easily and understand what the TAB function does. For example:

```
PRINT TAB(5);'RADIUS'
-----RADIUS

PRINT TAB(7.32);'FUN'
-------FUN
```

In the second example, because the expression is not integer, MINC uses only the integer portion — that is, the terminal spaces to column 8, skipping 7 spaces.

You should use a semicolon after the TAB function because a semicolon prevents MINC from skipping to the next print zone.

If the column number specified is less than or equal to the current column number, printing starts at the current position.

STRING VARIABLES

In many applications, it is necessary to input alphabetic characters. For example, suppose you want to use one of the graphic features of MINC to plot a set of data points. This program may first accept the labels of the axes (alphabetic data) as input and then the numeric data to plot. In this way, the program will work for any graph that needs to be produced.

For example, the following program accepts as input a name and address and prints them out. Remember that the backslashes let you put more than one statement on the same line.

```
10 PRINT 'YOUR NAME'; \ INPUT N$
20 PRINT 'YOUR ADDRESS'; \ INPUT S$,C$
30 PRINT \ PRINT N$ \ PRINT S$ \ PRINT C$
RUN
```

```
YOUR NAME? JOHN S. DOE
YOUR ADDRESS? 11 MAIN ST.
? WORCESTER MA. 01605

JOHN S. DOE
11 MAIN ST.
WORCESTER, MA. 01605
```

The variables N$, S$, and C$ are *string variables*. That is, their values must be strings. A string variable name is any variable name followed by a dollar sign ($). Examples of string variable names are:

```
A$
B$
D7$
```

The string variable A$ is a separate and distinct variable from the numeric variable A. Both names A$ and A can be used within the same program.

Input to string variables entered in response to the question mark from the INPUT statement need not be enclosed within quotation marks. MINC will accept input with or without quotation marks. However, if a comma is part of the input string, you must enclose the data within quotation marks or MINC will think you are entering more than one string. String variables can have as a value any collection of characters, spaces, and numbers. The maximum length of a string variable is 256 characters. For example,

```
10 PRINT 'input only one string variable'
20 INPUT A$
30 PRINT A$
RUN

NONAME              26-JUL-80           14:23:18

input only one string variable
? one, two
?MINC-W-Extra values from keyboard or file ignored at line 20
one

READY
```

Just as MINC sets numeric variables to zero when you run a program (until you give them a value), MINC sets string variables to the *null string* — the string with zero characters (the string constant " "). A string variable can take on the value of

```
10 LINPUT B$
20 PRINT B$
30 END
RUN

NONAME              09-MAY-80          09:50:47

? "Now, look here!", said John.
"Now, look here!", said John.

READY
```

The following program, which is identical to that above except that it inputs string data with an INPUT statement, prints a warning message. The comma after the quotation mark terminates the string. Notice that the quotation marks delimit the string but are not part of it.

```
10 INPUT B$
LIST

NONAME              09-MAY-80          09:52:15

10 INPUT B$
20 PRINT B$
30 END

READY
RUN

NONAME              09-MAY-80          09:52:18

? "Now, look here!", said John.
?MINC-W-Extra values from keyboard or file ignored at line 10
Now, look here!

READY
```

Concatenating Strings

The only string operation is concatenation. *Concatenation* puts one string after another without any intervening characters. Concatenation is specified by either the plus sign (+) or the ampersand (&). For example:

```
PRINT 'ONE' + 'WORD'

ONEWORD

READY

A$ = 'GOOD'&'BYE'
```

```
READY
PRINT A$

GOODBYE

READY
10 INPUT A$,B$,C$
20 PRINT A$ + B$&C$
RUN

NONAME                  09-MAY-80          10:23:22

? ONE,WO,RD
ONEWORD

READY
```

THE REMARK STATEMENT

Since BASIC does not allow very mnemonic variable names, and since programmers quite often forget the details of their programs, the REMARK statement allows a programmer to place comments within a program. These comments are ignored by MINC when your program is run. Comments in a program are valuable and should be used if someone else will use your program or if you will be looking at your program at a later date — long after you remember how you wrote the program. REM is a valid abbreviation for REMARK. For example:

```
10 REMARK THIS PROGRAM CONVERTS DEGREES
20 REM to radians
30 INPUT D
40 PRINT D*PI/180
RUN

NONAME                  09-MAY-80          10:27:35

? 90
  1.5708
```

Both forms of the REMARK statement, REMARK and REM, do the same thing.

MINC does not convert lower case characters to capitals in a REMARK statement. By using lower case characters in a REMARK statement, you can visually break up a program.

MINC stores REMARK statements in the workspace along with the other statements in the program. If a program gets too large

for the workspace, you can remove the REMARK statements to provide space. This will not affect the execution of the program.

The REM statement message itself can contain any printing character on the keyboard except the backslash, which will terminate the remark. Remarks are also terminated by a new line. For example:

```
10 REM THIS IS A "?@#! COMMENT
20 REM N$ represents the person's name
30 REM - A$ represents the person's address \INPUT A$
40 REM - C$ represents the person's city and state \INPUT C$
50 REM ** this is the end of this section **
```

CHAPTER 4
MINC PROGRAM COMMANDS

After the engineer wrote the program to compute the area of a circle and the volume of a cylinder, he wanted to save the program so he can use it repeatedly. This chapter describes the commands used for saving programs as well as those used for changing them.

Commands are those BASIC instructions that cannot be used with line numbers in the program mode. *Statements* are those BASIC instructions that can be used with line numbers in the program mode. In the immediate mode, statements look like commands.

For example, PRINT is a statement because you can put it into a program. RUN is a command, and cannot be put in a program.

CORRECTING PROGRAMS

Suppose the engineer had initially typed his program like this:

```
1 PRINT 'Radius'; \ INPUT R
2 PRINT 'Area'; PU*R^2
```

This program computes only the area of a circle. However, the engineer made a typographical error and typed PU instead of PI. He can easily correct this error by retyping the whole line as follows.

```
2 PRINT 'Area'; PI*R^2
```

Now that the engineer retyped line 2, MINC replaces the old line 2 in the workspace with this corrected one. The old line 2 is now gone from the workspace.

The general form of the SUB command is:

SUB stmt# d bad-string d good-string d whole-number

where:

stmt#	is the number of the line to be changed.
d	is any character to delimit the strings. The delimiter must not appear in either string. The bracket ([) character is a good choice for a delimiting character because it is not used in BASIC programs except in string constants.
bad-string	is the old series of characters to be deleted. Do not delimit the string with quotation marks.
good-string	is the new series of characters to be inserted. Do not delimit the string with quotation marks.
whole-number	is the constant that specifies which occurrence of bad-string in the line is to be replaced with good-string when there is more than one occurrence. Whole-number is optional and, if omitted, MINC replaces the first occurrence.

After you have made the changes to the line, MINC prints the new line.

Suppose you have the following incorrect line and want to replace the second F9 with I.

```
100 F9=A*SIN(X)+F9
```

Type the following SUB command to correct line 100.

```
SUB 100 [ F9 [ I [ 2
100 F9=A*SIN(X)+I

READY
```

You can use the SUB command to change a line number. For example, if the current line 20 is:

```
20 PRINT A,B,C,D
```

and you want to change the line number to 100, type the following SUB command:

```
SUB 20 [ 20 [ 100
100 PRINT A,B,C,D

READY
```

MINC makes the correction and then prints the new line. Since line 100 is a different line from line 20, MINC copies line 20 into line 100, but does not delete line 20. If you no longer want line 20, you can delete it with the DEL command (explained previously in this chapter).

You cannot use the SUB command to delete a line number (that is, change a program line to an immediate mode statement). If you try to do this, MINC prints the following message and does not execute the command.

```
?MINC-F-SUB creates an invalid statement or has a syntax error
```

MINC also prints this message if you entered the SUB command in the wrong format, such as omitting the delimiting character that ends the bad-string.

MINC uses the first character after the line number in the SUB command as the delimiting character, but it ignores all spaces and tabs until it finds a character. Consequently, you cannot use space or tab as a delimiting character. You also cannot use a digit as the delimiting character because MINC will consider it part of the line number.

If MINC cannot find the bad-string you specify, it reprints the line with no changes. Thus, you must type bad-string exactly as MINC lists it. To see how MINC lists a line, you use the LIST command explained later in this chapter.

Remember that MINC stores a program in upper case (except for string literals and REM statements), even though you typed it in lower case. Therefore, MINC would not find the following bad-string:

```
sub 2 [ pu [ pi
```

MINC has stored the typographical error, pu, as PU. Consequently, MINC would make no change in line 2. When you use

the SUB command, if you are not changing a string literal or a REM statement, it is a good idea to set the CAPS LOCK button so that you type in capital letters. Otherwise MINC will not find your bad-string in a SUB command (even though it recognizes the SUB command in lower case).

The Resequence Command

Suppose the engineer had initially typed his program like this:

```
1 PRINT 'Radius'; \ INPUT R
2 PRINT 'Area'; PI*R^2
```

If he wants to add height and volume calculations to his cylinder program, he must change the program. Since the statements are numbered consecutively, the engineer cannot fit a statement between lines 1 and 2 without renumbering the program.

He can solve his problem by typing the following resequence (RESEQ) command:

```
RESEQ
```

MINC renumbers the entire program starting with the first statement of the program, numbering it as 10 and the following statements in steps of 10. Thus, after the engineer types the RESEQ command, MINC changes his program to this:

```
10 PRINT 'Radius'; \ INPUT R
20 PRINT 'Area'; PI*R^2
```

You can use the RESEQ command to resequence your entire program or sections of your program. You can specify the new line number at which you want the renumbered section (or entire program) to start, the range of line numbers that you want resequenced (the old line numbers), and the increment to be used between each line number.

The general form of the resequence command is:

RESEQ newstart,oldstart-oldfinish,increment

As shown in the engineer's example, all of these values are optional:

newstart	specifies the new starting statement number.
oldstart	specifies the lowest existing statement number to be resequenced.

–oldfinish specifies the highest existing statement number to be resequenced. You must specify the hyphen before oldfinish.

increment specifies the increment to be used between each statement number.

If you do not specify an newstart, MINC uses the highest existing line number less than oldstart. For example, in the following RESEQ command, the newstart value is missing. If the highest line number under 105 is 100, MINC renumbers the first new line (line 105 or first highest above 105) to 100 plus the increment, or 110.

```
RESEQ ,105-200,10
```

In the above example, MINC renumbers old lines 105 through 200. If you omit the newstart value, you must leave in the comma before oldstart so that MINC knows that the number is oldstart and not newstart.

If you leave out oldstart, MINC starts resequencing at the beginning of the program. For example, the following RESEQ command resequences the program from the first line through old line 200 in increments of 10. If you leave out oldfinish instead, MINC resequences from oldstart to the end of the program.

```
RESEQ ,-200
```

If you do not specify an increment, MINC increments the statement numbers by 10.

For more details of the RESEQ command, see Book 3.

SAVING PROGRAMS

You can save any program that you type into the workspace for subsequent use. Once you have saved a program, you can do all of the following:

- Replace it. That is, change it and save the changed copy.
- Look at your directory. That is, look at a list of all the programs you saved.
- Delete a program. That is, throw a program away.

- Recover unused space. That is, consolidate all the empty space created by deleting programs.
- Print a program.
- Combine two programs.

The SAVE Command

Suppose you want to save the engineer's program, which you named CYLIND with the NEW command, and which you typed into the workspace. You can save this program named CYLIND by typing the following command.

```
READY
SAVE
```

After you have typed this SAVE command, the program is named CYLIND and saved on the volume in SY0: (the left drive).

When you type only:

```
SAVE
```

then MINC uses the default name in the workspace. Since you used the NEW command, the name in the workspace is CYLIND. If you had used the SCR command instead of the NEW command, the name in the workspace would be NONAME.

If you do not want your saved program to have the default name, use the following form of the SAVE command.

SAVE name

The specified name overrides the workspace name. For example:

```
SAVE CIRCLE
```

With this SAVE command, the program stored in the workspace is saved on volume SY0: with the name CIRCLE (no matter what the name of the workspace is). The program is also still in the workspace. For more details of the SAVE command, see Book 3.

Program Files

When you save a program, the program is stored on a volume in a *program file*. For example, if you type:

```
SAVE CYLIND
```

the program in the workspace is stored on SY0: in a program file called CYLIND.BAS. The .BAS denotes the type of the file named CYLIND. CYLIND.BAS is a file that holds a BASIC program.

The full *file specification* for the CYLIND program is:

```
SY0:CYLIND.BAS
```

where:

SY0:	is the device on whose volume the program is stored. If you leave out the device, MINC defaults to SY0:. Note that you must always have a colon as part of the device name.
CYLIND	is the program name. You can select any name between one and six characters in length.
.BAS	is the file type denoting that CYLIND is a file that contains a BASIC program. The .BAS file type is the default file type for BASIC programs.

You can store a program in a file with any file type (up to three characters) you choose. However, if you make up your own file type, you must always enter the file type, or MINC will not find the correct file.

For example, you can enter:

```
SAVE CYLIND.PRG
```

Now, whenever you want to reference this program (to run it, for example) you must always use the name and file type, CYLIND.PRG. If you use the default file type, you need only to type the name.

For example, if you want to run the program stored in CYLIND.PRG, you must type:

```
RUN CYLIND.PRG
```

However, if you want to run the program stored in CYLIND.BAS, type:

```
RUN CYLIND
```

Below are other examples:

SAVE	saves the program currently in the workspace in a file called SY0:NONAME.BAS, unless you have used the NEW command. If you have used the NEW command to name the workspace, SAVE saves the program with the name of the workspace.
SAVE FUN	saves the program currently in the workspace in a file called SY0:FUN.BAS.
SAVE SY0:FUN.PRG	saves the program currently in the workspace in a file called SY0:FUN.PRG. Remember that you must always type the file type when it is other than the default.
SAVE SY0:HELLO.BAS	saves SY0:HELLO.BAS.

Later in this chapter, saving programs on SY1: will be discussed.

The general form of the SAVE command is:

SAVE filespec

where filespec takes the form:

dev:name.typ

and

dev: is the device (default SY0:)

name is the program file name (default NONAME)

.typ is the file type (default .BAS)

There are types of files other than program files. For example, if you wanted to collect data from an instrument, you would save

this data in a *data file*. You cannot run data files or any file that is not a program file. These types of files are discussed in Chapters 11, 14, and 15.

If you try to save a program with the same name as another program already stored on the volume, MINC prints an error message. For example, if you have already saved the program CYLIND.BAS and you want to correct the program and save it again, you cannot type:

```
SAVE CYLIND
```

MINC prints the following error message.

```
?MINC-F-File name in use; REPLACE or change name or volume
```

You cannot use the SAVE command to store the new version of CYLIND. You must use the REPLACE command (described in one of the following sections) or change the file description (the volume, name, or file type).

The OLD Command

The OLD command brings a saved program from the program file on a volume into the workspace. For example, the following OLD command brings the CYLIND program stored in SY0:CYLIND.BAS into the workspace.

```
READY
OLD
OLD FILE NAME -- CYLIND

READY
LIST

CYLIND                  13-APR-80               13:46:43

10 PRINT 'Radius'; \ INPUT R
20 PRINT 'Height'; \ INPUT H
30 PRINT 'Area';PI*R^2
40 PRINT 'Volume'; \ PI*R^2*H

READY
```

Like the SAVE command, the OLD command uses SY0: as the default device and .BAS as the default file type.

Notice in the above example, if you type only OLD, MINC prompts you for the OLD FILE NAME. You can also type the file name as part of the OLD command. For example:

OLD filespec

where filespec is of the form:

dev:name.typ

The OLD command first changes the workspace name, then erases the workspace (performs a SCR), and finally brings in the program stored in the program file named by the filespec.

If there is no program file with the name specified in filespec, MINC prints the following error message:

```
?MINC-F-Specified or default volume does not have file named
```

When this error occurs, the workspace name changes, but MINC does not scratch the workspace. Thus, you must be careful not to inadvertently change a program name through this sort of error.

The REPLACE Command

If you save a program, then call it back into the workspace with an OLD command, and then alter the program, you cannot use the SAVE command to put the changed program back into the old file. You can save this changed program with a new name, or you can use the REPLACE command.

The REPLACE command is like the SAVE command except that REPLACE saves a program even if it means writing over an existing program file. This difference between the SAVE and REPLACE commands helps prevent you from inadvertently deleting program files that you previously saved.

The form of the REPLACE command is:

REPLACE filespec

If you omit the file specification, MINC uses the current workspace name as the default specification.

Study the following example and notice the difference between SAVE and REPLACE. This example uses the name SY0:TEST.BAS throughout (established with the NEW command).

```
NEW
NEW FILE NAME -- TEST

READY
```

```
10 PRINT 'This is a test'
SAVE

READY
20 PRINT 'This is another test'
LIST

TEST                    09-MAY-80              00:03:43

10 PRINT 'This is a test'
20 PRINT 'This is another test'

READY
SAVE
?MINC-F-File name in use; REPLACE or change name or volume

READY
REPLACE

READY
```

Now the program stored on the diskette in file TEST.BAS is the new version with two lines.

For more details of the REPLACE command, see Book 3.

The DIRECTORY Command

You can see the names of all the program files and other types of files that you save on a diskette with the DIRECTORY command (DIR).

On each diskette, MINC keeps a directory on each diskette of all the files stored on that diskette. For example, suppose you have saved two programs named CYLIND.BAS and SINES.BAS and you want to look at the directory. You can use the DIR command as shown below:

```
READY
DIR

10-MAY-80
Volume ID:  MINC System
Owner:      engineer
SINES.BAS  1      11-MAY-80      CYLIND.BAS      1      12-MAY-80
<unused>   122
2 Files, 2 Blocks
122 Free Blocks

READY
```

The DIR command causes MINC to print out the directory,

which consists of the current date, the volume identifier, the owner, the list of files, and a description of the available and used space. All of these features are described below.

The *Volume ID* and *Owner* were named by you or someone else when the diskette was initialized (described later in this chapter under *Initializing a Volume).*

The list of program files takes the form:

name.typ b dd-mmm-yy name.typ b dd-mmm-yy

where:

name.typ	is the name of the file and its file type.
b	is the size of the program file in blocks. A block is a unit of size on a diskette. Blocks and diskettes are discussed more thoroughly later in this chapter (in the section titled MINC Volumes).
dd-mmm-yy	is the date that the program was saved or replaced.

The <unused> shows you the unused, available space on the diskette.

The last part of the directory listing takes the form:

n Files, m Blocks
x Free Blocks

where:

n Files	is the number of files in the directory.
m Blocks	is the total number of blocks that the n programs take.
x Free Blocks	is the total available space on the diskette.

There are a number of features of the DIR command. With the DIR command you can:

- Look at a directory of all files on a diskette.

- Look at a directory of all file names on a diskette that have the same file type; for example, all file names with the .BAS type.
- Look at a directory of all files with the same name but different file types.
- Direct the directory listing to another file.
- Direct the directory listing to the line printer.

For more details of these advanced features of the DIR command, see the DIRECTORY command in Book 3.

Printing a Program

Until now, you have been using the LIST command to see the program that is stored in the workspace. You cannot use the LIST command to view the contents of a file stored on a volume.

To see a listing of the contents of a file that is not in the workspace on the terminal screen, use the TYPE command. The form of the TYPE command is:

TYPE filespec

where filespec takes the form:

dev:name.typ

If you leave out the device, MINC defaults to SY0:. If you leave out the file type, MINC defaults to a program file (.BAS file type). You can display files other than program files by specifying the name and file type. (For more about files other than program files, see Chapter 11.)

The TYPE command does not affect the use of the workspace. It reads the file from the volume and displays it without storing it in the workspace. That is, if you have a program in the workspace and you command MINC to type another program, the original program in the workspace remains unaffected.

For example:

```
READY
TYPE CYLIND
10 PRINT 'Radius'; \ INPUT R
20 PRINT 'Height'; \ INPUT H
30 PRINT 'Area';PI*R^2
40 PRINT 'Volume'; \ PI*R^2*H
```

```
READY
```

You can use the NO SCROLL key to stop a long program from scrolling off the top of the screen before you have a chance to read it.

For more details of the TYPE command, see Book 3.

Checking the Length of a Program

The LENGTH command lets you check to see how much workspace a program requires. The length of the program is given in units of workspace called *words*.

The form of the LENGTH command is:

```
LENGTH
```

To see how many words are available in your workspace, type the LENGTH command after the SCR command.

```
READY
SCR

READY
LENGTH

0 USED, 5491 FREE

READY
```

In the following example, when the CYLIND program is in the workspace, it uses 58 words (58 USED) and leaves 5433 words available (5433 FREE).

```
CYLIND                11-MAY-80              11:22:06

10 PRINT 'Radius'; \ INPUT R
20 PRINT 'Height'; \ INPUT H
30 PRINT 'Area';PI*R^2
40 PRINT 'Volume'; \ PI*R^2*H

READY
LENGTH

58 USED, 5433 FREE
```

The length of a program can vary from run to run. For example, if one of your programs accepts string input, the length of the

The LISTNH command is the same as the LIST command except that MINC displays no header (NH).

The following example lists the entire program without a header line.

```
LISTNH
10 PRINT 'Radius'; \ Input R
20 PRINT 'Height'; \ Input H
30 PRINT 'Area';PI*R^2
40 PRINT 'Volume'; \ PI*R^2*H
```

This next LIST command lists the header line and statement number 20 of the program.

```
LIST 20

CYLIND          11-MAY-80          11:22:10

20 PRINT 'Height'; \ INPUT H
```

It is often helpful to list a line before using the SUB command. By listing the line, you can see exactly what it looks like so you can correctly change the line.

This last example of the LIST command lists statements 25 and 50, all statements from 100 through 200, and all lines from statement 500 through the end of the program.

```
LIST 25,50,100-200,500-
```

Running Programs

You can run a program that is stored in the workspace by typing:

```
RUN
```

When MINC executes the RUN command, it first displays a header line. It then initializes all numeric variables to 0 and all string variables to the null string. Finally MINC starts executing the program at the lowest numbered line.

The RUNNH command has the same effect as the RUN command except that MINC does not print the header line.

It is a good practice to use the RUN command rather than the RUNNH command. In case your program has some sort of error, you at least see the header and know the program tried to run.

If you want to run a program from a program file, that is, a program stored on the diskette but not in the workspace, type:

RUN filespec

where filespec takes the form:

dev:name.typ

If you do not specify the device, MINC defaults to SY0:. If you do not specify the file type, MINC defaults to the program file type (.BAS).

When you specify a file with the RUN command, MINC erases the workspace (equivalent to SCR), changes the workspace name, and brings the specified program into the workspace. Finally MINC starts execution of the program. This process destroys the previous contents of the workspace, so be sure to save it if you want to keep it before executing the RUN filespec command.

The RUN filespec command does not display a header line and is equivalent to the RUNNH filespec command. It leaves the program specified by filespec in the workspace when it is finished.

For example, the following sequence of commands brings the CYLIND program into the workspace and runs the program.

```
READY
RUN CYLIND

CYLIND          11-MAY-80          12:10:08

Radius?
```

If MINC cannot find the file specified in a RUN filespec command, it changes the workspace name and displays the following error message.

```
?MINC-F-Specified or default volume does not have file named
```

However, MINC does not delete the program originally in the workspace. This can happen if you mistype the RUNNH command, such as RUNHN. In this case, MINC interprets the command as RUN HN, where HN is the file specification.

If the file specification begins with NH, MINC assumes that the

NH is part of the RUNNH command. For example, MINC interprets a RUN NHTEST command as RUNNH TEST. To run a program whose file specification begins with NH, use the RUNNH command.

COMPILING A PROGRAM

MINC does not store a program in the workspace exactly the way you type it but instead compresses (compiles) each line. By compiling the program internally, MINC allows you to fit larger programs in the workspace than you could if MINC did not compile each line.

Whenever you list the program or save it on a diskette, MINC translates the program from the compiled form to the form that you entered.

The COMPILE command saves the internal, compiled version on a diskette. The BASIC version is still in the workspace. Once you have saved this compiled version, MINC can bring this version into the workspace faster than it can bring a version saved with the SAVE command into the workspace. Thus the OLD and RUN filespec commands work faster.

The form of the COMPILE command is:

COMPILE filespec

Where filespec takes the form:

dev:name.typ

If you omit the filespec, MINC uses the current workspace name with the .BAC file type, which stands for a compiled program. If you omit the device, MINC defaults to SY0:. If you specify the filespec, MINC stores the compiled version into that file on the diskette.

You can COMPILE only the program currently stored in the workspace.

If you type the following command:

RUN name

MINC tries to run name.BAC first. If name.BAC is not on the diskette, then MINC tries to run name.BAS.

You can use the TYPE command to type a compiled program. However, the output is unintelligible and quite confusing.

USING OTHER VOLUMES

So far you have saved programs and run programs; all of these programs have been stored on the diskette in the SY0: device. You can use both SY0: and SY1: with MINC, but there are some general procedures you must follow to prepare a new diskette for use.

In general, to prepare a new volume (diskette) for use, you must:

- Place the new volume in SY1:
- Initialize the volume (set up an empty directory).

After you have initialized the volume, you can copy programs, save programs, and store data on it.

The following sections discuss MINC volumes, their structure, and their use.

MINC Volumes

The volumes that you have been using up to now are MINC *system volumes*. That is, these volumes have system files on them that make MINC work. When you put these system volumes in SY0: (the left drive) and turn on the power, they let you know that they are system volumes by printing one of the following messages:

```
MINC BASIC V1.2 for the 11/23
MINC BASIC V1.2 for the 11/03
```

Not all volumes are system volumes. You can use a volume to store programs you have written or to store data that you have collected. If you try to start MINC with one of these *nonsystem volumes* in SY0:, you will get an error message similar to the following:

```
@
?BOOT-F-No boot on Volume
```

When you get such a message, you must put a system volume in SY0: and start again.

You must always have a MINC system volume in SY0: or your MINC system will not work.

The following sections describe volumes in more detail.

Master Volumes With your MINC system you received Master volumes. They are a form of MINC system volume. You cannot use these volumes because they have been set up so that you can

only copy them. They direct you to place an empty volume in SY1: and then they copy the MINC system onto your volume. For more information, see Book 1.

You cannot manipulate a Master volume. That is, you cannot save programs or data on a Master volume. To save programs or data, you must copy the Master volume onto your own volume or volumes and then use your own volume.

You should keep many copies of the MINC system volume. That way, in case one volume is somehow damaged, you have another copy. The procedure for copying volumes is described in the following sections.

The Structure of Volumes A volume is a storage area for MINC. A volume has a finite amount of space; that is, it can hold only a finite amount of information.

The unit of size on a volume is a *block*, which holds 512 characters.

Figure 1 represents the organization of a new diskette. Notice that nothing is stored on it.

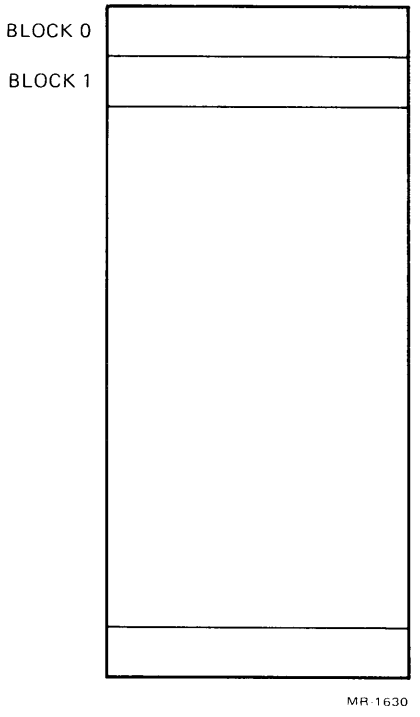


Figure 1. Organization of a New Diskette

Figure 2 represents the organization of an initialized diskette. When you *initialize* a diskette, MINC sets up a directory on that diskette. Nothing is stored in the directory, however, until you begin using the diskette for program storage or to copy system files. (The procedure for initializing a diskette is described later in this chapter.)

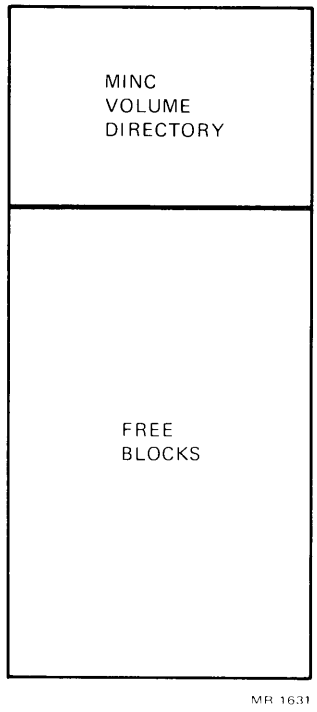


Figure 2. Initialized Diskette with No Files

Figure 3 represents a diskette that stores only program files. After initializing a new diskette, you can place it in SY1: and type the following SAVE commands.

```
READY
OLD SINES

READY
SAVE SY1:SINES.BAS

READY
OLD CYLIND

READY
SAVE SY1:CYLIND.BAS

READY
```

Now the diskette in SY1: holds only two program files and organizationally looks like Figure 3.

Figure 4 represents a system volume on which you saved the SINES program and CYLIND program on SY0:.

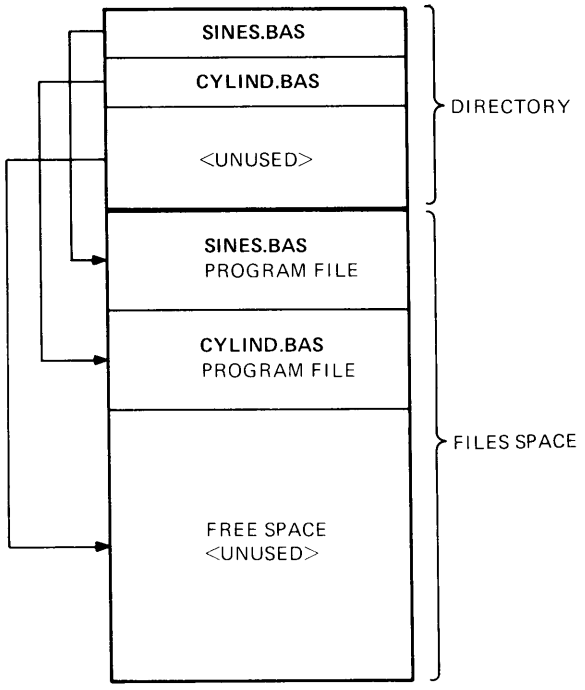


Figure 3. Diskette with Program Files

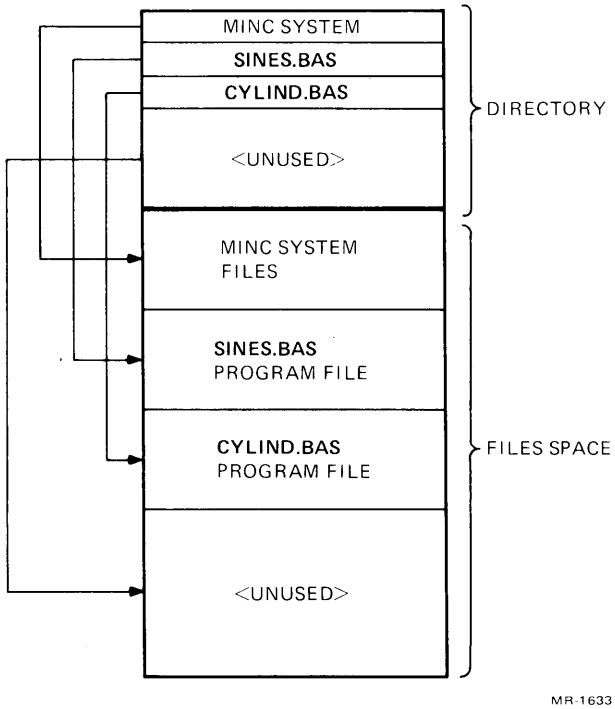


Figure 4. MINC System Volume with Program Files

When you look at the directory of a system volume, you cannot see the MINC system files; however, they are on the volume. Thus, the directory of the volume in Figure 3 would look the same on the screen as the directory of the volume in Figure 4. The only difference in the directory listings is that the number of free blocks is less for the system volume than for the nonsystem volume. After all, the MINC system files use up some of the space on the volume.

Deleting a Program File

You can delete any file from a diskette with the UNSAVE command. The form of the UNSAVE command is:

UNSAVE filespec

where filespec takes the form:

dev:name.typ

If you leave out the device, MINC defaults to SY0:. If you leave out the file type, MINC defaults to the program file type (.BAS). You can unsave files other than program files by specifying the name and file type. (For more about files other than program files, see Chapter 11.)

For example:

```
UNSAVE SY0:SINES.BAS
```

Files take up space (blocks) on a diskette, and eventually they can fill it. Use the UNSAVE command to delete files that you no longer need to store, to make more space.

When you unsave a file, MINC takes the file name out of the directory, and the space where the file was stored becomes immediately available. Figure 5 shows the organization of a system diskette with an unsaved file.

Recovering Unused Space

When you unsave a file, the space where the file was stored becomes immediately available. However, if you look at Figure 5, the SINES.BAS file was between the MINC system files and the CYLIND.BAS file. Thus, where the SINES.BAS file was deleted, it left a "hole" in the directory and in the available space. Only programs of the same size or smaller will fit in the holes.

To consolidate all of the unused space into one large area, use the COLLECT command.

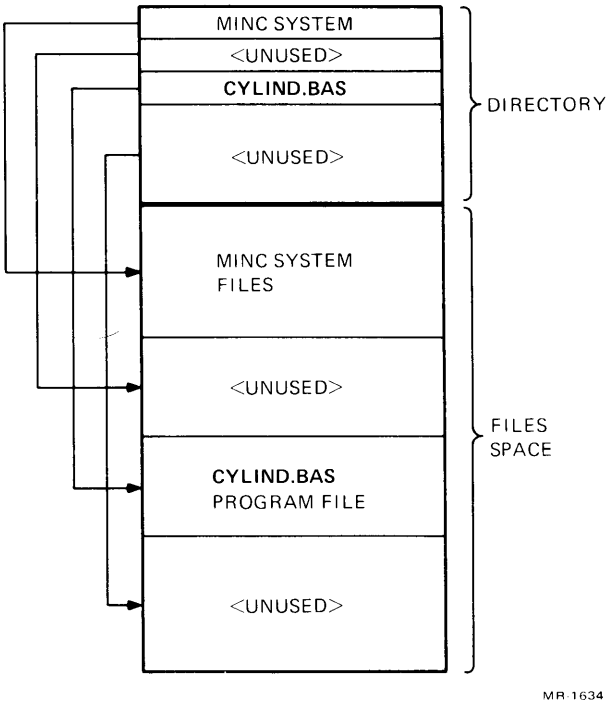


Figure 5. Diskette with File Deleted

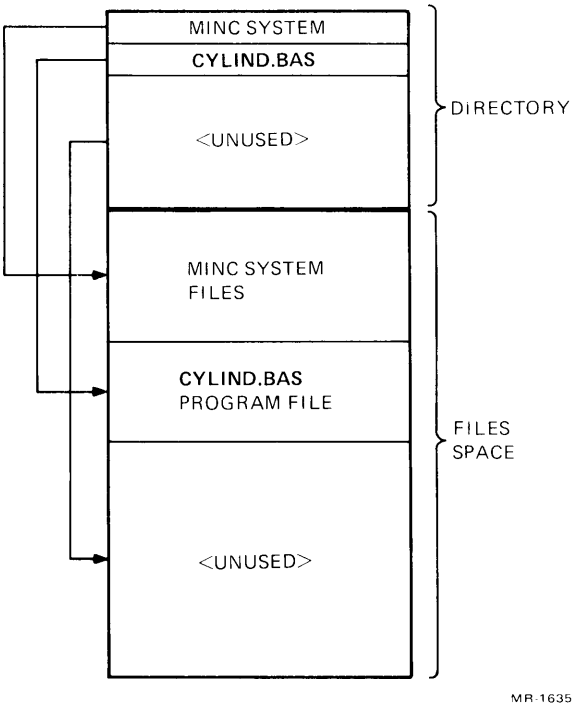


Figure 6. Collected Diskette

The form of the COLLECT command is:

COLLECT dev:

where dev: is either SY0: or SY1:.

If you collect the diskette shown in Figure 5, graphically it will look like Figure 6.

You can also use COL instead of COLLECT. COL is a valid abbreviation for COLLECT.

NOTE

Before you collect a diskette, you should verify it for bad blocks. This procedure is explained later in this chapter.

Initializing a Volume

Before using a new volume, you must initialize it. When you initialize a volume, MINC formats it — setting up a directory area and an area available for files. Besides putting the volume in the necessary format, initializing it also erases any previous information stored on the volume.

To initialize a volume, make sure you have a system volume in SY0: and the volume to be initialized in SY1:. Then type one of the following commands:

```
INI SY1:

INITIALIZE SY1:
```

MINC responds with:

```
Install volume to be initialized in SY1:, and press RETURN

  Current volume id:eeeeeeeeeeee
  Current owner:    eeeeeeeeeeee
  Proceed with initialization (Y or N)?
```

If the volume is new (that is, never used), the *e*'s appear as the "Current volume id" and the "Current owner". If the volume has been used before, the "Current volume id" and the "Current owner" are those names given in the previous initialization. The volume id and owner are a means of ensuring that you are not initializing a valid volume (that is, one with valid information on it).

If you have put a valid volume in SY1: and do not want to proceed with the initialization, type N for no. If you want to proceed, type Y for yes. If you type anything but Y or N, MINC uses N as the default answer and does not initialize the volume.

If you want to proceed, type Y for yes at the question mark. MINC then prompts you for the following information.

```
Type new Volume id:
```

The id stands for identifier. Type the identification name that you wish to give the volume. It is recommended that you give the volume a meaningful identifier. A maximum of 12 letters is allowed for the volume identifier. If this volume will be a system volume, you should denote it as such in the volume id. For example, if you are copying one of the Master diskettes you might want to name the copy "mastersystem," and then make more copies of the "mastersystem" diskette for general use. You can answer the question as follows:

```
Type new Volume id: Lab 3 system
```

Then, the next question is as follows.

```
Type new owner name:
```

If your name is Jane Doe and this volume is your personal volume, you might want to type Jane Doe as the owner name.

Then MINC will initialize the volume. This process takes one or two minutes. Finally, MINC prints the message:

```
Initialization is complete; found xxx Bad blocks
```

MINC does not actually type "xxx". It types a 3-digit number representing the number of *bad blocks*. Remember a block is a unit of diskette space. A bad block is a block that MINC cannot use because it has been damaged in some way. Most diskettes do not have bad blocks. You can limit the occurrence of bad blocks by taking good care of your diskettes. However, with normal use, a diskette can develop a bad block after a long period of time through no fault of your own. See Book 1 for the instructions on caring for a diskette.

It is possible to use a volume that has bad blocks. However, if you are a beginner, put volumes with bad blocks aside for now, because the procedure for using them can be more difficult. For

sage similar to the following example. The only column that gives you useful information is the filename column. In this example, both of the bad blocks are marked in the FILE.BAD file. Thus, they will not affect your use of the diskette.

```
Bad Blocks      Type        Filename        Rel Blk

    414         Hard        FILE.BAD          0
    417         Hard        FILE.BAD          0

READY
```

If one of your diskettes develops bad blocks in one of your files, your file name will appear under the Filename column. For example, if bad blocks develop in CYLIND.BAS, you can recover the file. See the Error Recovery section in Book 3.

If one of your diskettes develops bad blocks in one of the MINC system files, the recovery procedure can be more difficult. See the Error Recovery in Book 3.

Duplicating a Volume

You can use the DUP command to copy an entire diskette — including the MINC system files. The form of the duplicate command is:

DUP

You should use the DUP command to keep backup copies of each of your diskettes. You should do this often, at least once a day, because diskettes are fragile.

Use the INI command to prepare a new diskette before using it for duplicating.

The DUP command prompts you to put the diskette to be duplicated in SY0: and the backup diskette in SY1:. The DUP command then copies the entire contents of the diskette in SY0: to the diskette in SY1:, destroying all of the previous contents of the diskette in SY1:.

If you are going to use a used diskette as a backup, you must initialize it first to mark any bad blocks that may have developed since the last initialization. The DUPLICATE command prints the following message if your diskette has not been recently initialized.

```
?UTILITY-F-Target volume must be newly initialized
```

The following example shows the entire procedure for duplicating a diskette.

```
INI SY1:
Install volume to be initialized in SY1:, and press RETURN

Current volume id:eeeeeeeeeeee
Current owner:    eeeeeeeeeeee
Proceed with initialization (Y or N)?Y
Type new Volume id:MINC system
Type new owner name:Student 003

Initialization is complete; found 000 Bad blocks

READY
verify sy1:
There were no bad blocks found

READY
DUP
Install volume to be duplicated in SY0:;

Install initialized, empty volume in SY1:, are you ready (Y or N)?Y

SY1 volume id is: MINC system
SY1 owner is:     Student 003

Do you want to duplicate another volume (Y or N)?N

Re-install system volume in SY0:, and then press RETURN

MINC BASIC V1.2 for the 11/23 (or 11/03)

10-MAY-80
04:33:40

READY
```

The DUP command will even duplicate a diskette with bad blocks. Thus, if one of your files develops a bad block, you can recover the file and restore it to good condition on another diskette. For more information on this procedure, see DUPLICATE in Book 3.

Copying a File

You can use the COPY command to transfer a file from one volume to another or to list the file on a line printer.

The form of the COPY command is:

COPY from-filespec to-filespec

where:

from-filespec	is the file specification of the file to be copied. The default file type is .BAS. The name of the file to be copied must be present.
to-filespec	is the file specification of the new file that holds the copy. The default file name is the from-file name. The default file type is the from-file type. The default device is SY0:. If you use LP:, MINC prints the file on the line printer.

You must specify at least partially both the from-filespec and the to-filespec; that is, specify at least the file name if not the device or file type.

For example, either of the following COPY commands copy CYLIND.BAS from SY0: to SY1: as the program CYLIND.BAS.

```
COPY SY0:CYLIND.BAS SY1:CYLIND.BAS

COPY CYLIND SY1:
```

The COPY command copies one file at a time.

If you try to copy a file to a filespec that already exists, MINC displays the following message:

```
Output file name is already in use;
do you want to erase the current contents (Y or N)?
```

If you type N, MINC does nothing and returns to the READY prompt. If you type Y, MINC performs the copy, destroying the previous contents of the file.

CHAPTER 5
PROGRAM CONTROL

Until now, all of the programs given in examples have proceeded sequentially. That is, MINC executes each of the programs step by step in order by statement number until the last statement is executed, and then terminates the program. For example, the following program executes statement 20, then 30, then 40, then 50, and then, since there are no more statements, the program stops executing.

```
20 PRINT 'Radius'; \ INPUT R
30 PRINT 'Height'; \ INPUT H
40 PRINT 'Area'; PI*R^2
50 PRINT 'Volume'; PI*R^2*H
```

This chapter explains how you can gain more control over the order in which MINC executes statements and how to make some statements execute more than once.

IF STATEMENTS AND LOGICAL EXPRESSIONS

IF statements are one method by which you can control the flow of your program. The *flow* of a program is the order in which the statements are executed. In all previous examples, the flow has been sequential — that is, MINC starts executing statements with the lowest statement number and proceeds to those with the highest.

IF Statements in BASIC Are Like English

In English, the statement, "If it is sunny, then wash the car," is a conditional sentence: if the events in the first clause occur ("if it is sunny"), the events in the second clause should follow.

A more complicated if statement is: "If it is sunny, then wash the car; otherwise stay in the house."

BASIC has comparable forms of IF statements. One form of the IF statement (comparable to the simpler form in English) is demonstrated below.

```
IF A<B THEN A=A+1
```

In English this statement means, "If A is less than B, then increment A by 1"; and that is exactly what happens in BASIC.

The IF statement has three slightly different forms:

```
IF logical-expression THEN statement
IF logical-expression THEN stmt#
IF logical-expression GO TO stmt#
```

All of these forms of the IF statement are described in the following sections.

Later, programming the "otherwise" clause of an IF statement is explained.

NOTE

Do not use IF statements in multiple-statement lines. The outcome of trying to put more than one statement on the same line as an IF statement is not obvious and can actually be quite confusing. See IF statements in Book 3.

Logical Expressions

A *logical expression* is an expression that can take on one of two values — either the expression is true or it is false. In the previous example, "it is sunny" is a logical expression because it is either true or false. In BASIC, "A is less than B" is likewise a logical expression.

The form of a logical expression is:

```
expression1 relational-operator expression2
```

where expression1 and expression2 are both arithmetic expressions or both string expressions as defined earlier.

Relational operator can be one of six operators that work on expressions and are defined in the following tables.

Arithmetic Relational Operators Table 2 shows and explains the six arithmetic relational operators.

Table 2. The Arithmetic Relational Operators

Mathematical Symbol	*BASIC Symbol*	*Example*	*Meaning*
=	=	A=B	A is equal to B.
<	<	A<B	A is less than B.
≤	<=	A<=B	A is less than or equal to B.
>	>	A>B	A is greater than B.
≥	>=	A>=B	A is greater than or equal to B.
≠	<>	A<>B	A is not equal to B.

The symbols =<, =>,>< are accepted by BASIC but are converted to <=, >=, and <> and are shown in that form in a listing.

The arithmetic relations are the same in BASIC as in mathematics.

The expressions in a logical expression can, of course, be more than one numeric variable. For example:

```
(A+6*X-37)/43 < (C+19)^2
```

Whether this expression is true or false depends on the values of the variables in the expression. The expression could be false at the beginning of a program, and true later, after the values of the variables had changed.

String Relational Operators The same six relational operators that apply to arithmetic expressions also apply to string expressions. In string expressions, however, these relations apply to alphabetical sequence. The comparison is done character by character, left to right, on the internal representation of a character (called the *ASCII code*). For example, it is true that:

"ABC" < "ABD"

because "ABC" alphabetically precedes "ABD". MINC compares the first character from each expression; if they are equal, then MINC compares the second characters. This process continues until a character differs or until the strings match.

Because strings can also contain numbers and punctuation, each character, whether it is alphabetic, numeric, or other, has an ASCII code. The characters have a *collating sequence* that tells the "alphabetical order" of all the characters depending on

the value of each character's ASCII code. The collating sequence and the ASCII codes are given in Appendix A.

For example, it is true that:

"ABC" < "abc"

because the ASCII codes for the capital letters are lower than the ASCII codes for the lower case letters.

In any string comparison, MINC ignores trailing blanks (that is, "ABC" is equivalent to "ABC "). However, MINC does not ignore preceding blanks. Table 3 shows and explains the string relational operators.

Table 3. The String Relational Operators

Operator	*Example*	*Meaning*
=	A$=B$	The strings A$ and B$ are alphabetically equal.
<	A$<B$	The string A$ alphabetically precedes B$.
>	A$>B$	The string A$ alphabetically follows B$.
<=	A$<=B$	The string A$ is equivalent to or precedes B$ in alphabetical sequence.
>=	A$>=B$	The string A$ is equivalent to or follows B$ in alphabetical sequence.
<>	A$<>B$	The strings A$ and B$ are not alphabetically equal.

IF/THEN Statements

One form of the IF/THEN statement was demonstrated in the first section of this chapter. The two forms of the IF/THEN statement are:

IF logical-expression THEN statement
IF logical-expression THEN stmt#

Examples of the first form are:

```
10 IF B<=23 THEN B=B+1

10 IF B^2-4*A*C < 0 THEN PRINT 'ROOTS IMAGINARY'

10 IF A$<>B$ THEN F=1

10 IF C$>B$ THEN IF A$=B$ THEN S$=C$
```

In the first example, if it is true that B is less than or equal to 23, then B is incremented by one. If B is greater than 23, then MINC passes on to the next statement in the program (which is not listed here). In the final example, a second IF statement follows the THEN statement, showing that IF statements can be nested.

A program example using this form of the IF statement follows:

```
10 REM* This program prints the square root
12 REM* Of positive inputs. It flags
14 REM* Negative values.
16 REM*
20 PRINT 'Number';\INPUT N
30 IF N<0 THEN PRINT 'Square root of negative # not allowed'
40 IF N>=0 THEN PRINT 'The square root is'; SQR(N)
RUN

SQRT                    20-APR-80               4:15:27

Number? 4
The square root is 2

READY
RUN

SQRT                    20-APR-80               4:15:10

Number? -2
Square root of negative # not allowed
```

Examples of the second form are:

```
100 IF A$<B$ THEN 50

100 IF X+3>=18 THEN 200

100 IF 52<=X THEN 25
```

The second form of the IF statement says, "If the logical expression is true, then *transfer control* to the statement whose number is after the THEN." MINC has control over the execution of the program. IF statements cause MINC to transfer control from the IF statement to the statement whose number is after the THEN. In the first example, if A$ is less than B$, then the next statement executed will be statement 50. If A$ is not less than B$, then the next statement executed will be the statement that immediately follows line 100.

A program example using the second form of the IF statement is:

```
10 REM - This program sorts 3 input strings in
15 REM - alphabetical order.
20 PRINT 'Input 3 strings'
30 INPUT A$, B$, C$
60 REM - test if they are in alphabetical order.
70 IF A$<B$ THEN 110
75 REM - swap them.
80 S$=A$
90 A$=B$
100 B$=S$
110 IF A$<C$ THEN 150
115 REM - swap them.
120 S$=A$
130 A$=C$
140 C$=S$
150 IF B$<C$ THEN 190
155 REM - swap them.
160 S$=B$
170 B$=C$
180 C$=S$
190 PRINT 'The sorted list is'
195 PRINT A$
200 PRINT B$
210 PRINT C$
RUNNH

Input 3 strings
? hello
? goodbye
? always
The sorted list is
always
goodbye
hello
```

Notice that a third "temporary" variable is needed for a swap. You cannot swap A$ and B$ by saying:

```
A$=B$\B$=A$
```

because of the nature of an assignment statement.

```
A$=B$
```

*erases the previous value of A$* and puts the value of B$ there. Consequently, the original value of A$ is lost. In this example, S$ saves the original value of A$ while B$ is stored in A$.

IF/GO TO Statements

The form of the IF/GO TO statement is:

IF logical-expression GO TO stmt#

and MINC executes it exactly like the IF logical-expression THEN stmt# statement. The previous example that prints out three strings in alphabetical order would work the same if you replace THEN with GO TO. GO TO is perhaps clearer in this type of IF statement because the term GO TO itself implies transfer of program control.

Programming the "Otherwise"

The sorting example program is actually an example of programming the "otherwise" portion of an IF statement. For example, the following program segment says, "If A$ is less than C$ then compare B$ and C$; otherwise swap A$ and C$."

```
110 IF A$<C$ GO TO 150
115 REM - - - otherwise swap A$ and C$
120 S$=A$
130 A$=C$
140 C$=S$
150 IF B$<C$ THEN . . .
.
.
.
```

Statements 120, 130, and 140 are all part of the "otherwise."

GO TO STATEMENTS

GO TO statements can be used anywhere in a BASIC program to transfer control to another statement.

Unconditional GO TO Statements

Upon execution of an unconditional GO TO statement, MINC transfers control to the statement-number in the GO TO statement. This form of GO TO statement is called *unconditional* because control is always transferred when the statement is executed, as opposed to the IF/GO TO statement, which is a conditional GO TO. The form of the unconditional GO TO is:

GO TO stmt#

A program using an unconditional GO TO was shown in Chapter 3 for computing sines and cosines. The following statement numbers are the same as those used in Chapter 3. Some REMARK statements have been added to help you understand the program.

```
5 REM - This program computes sines and cosines
10 REM - of any angle given in degrees.
20 PRINT ,'Sine', 'Cosine'
30 PRINT 'Degrees';
35 REM - Input angle in degrees.
40 INPUT X
55 REM - Compute radians from angle.
60 Z=X*PI/180
70 PRINT ,SIN(Z), COS(Z)
75 REM - Repeat for next angle
80 GO TO 30
```

At statement 80, control always passes back to statement 30 because of the unconditional GO TO.

ON/GO TO Statements

Unconditional GO TO statements transfer control to a particular statement. ON/GO TO statements transfer control to one of several statements depending upon the value of an expression. Thus, ON/GO TO statements are a form of conditional GO TO. The other form of a conditional GO TO is the IF/GO TO statement.

The form of the ON/GO TO statement is:

ON numeric-expression GO TO list-of-line-numbers

For example:

```
10 ON X GO TO 100,200,300
```

If X is equal to 1, control transfers to statement 100. If X is equal to 2, control transfers to statement 200. And, if X is equal to 3, control transfers to line 300. If X is less than 1 or greater than 3, then MINC prints the following error message.

```
?MINC-F-Value of control expression is out of range at line 10
```

Line 10 is the line number with the ON/GO TO statement.

An example of using ON/GO TO statements is given in Chapter 7.

RESEQUENCING PROGRAMS WITH GO TOs

When you use the RESEQ command to renumber your statements, MINC also renumbers any references to these statements. That is, all line numbers referenced in GO TO, ON/GO TO, IF/THEN, and IF/GO TO statements are changed to reflect the new numbering. However, MINC does not change any references to statement numbers within remarks.

Note that if you change statement numbers yourself, you must be careful to change any references to these statements.

PROGRAM TERMINATION

Obviously, every program must terminate at some time. Programs can terminate in any of the following ways:

- Normally, by executing the highest numbered statement.
- Normally, by executing a STOP or END command (explained in the following sections).
- Abnormally, by encountering a fatal error (denoted by messages beginning with ?MINC-F-).
- Normally or abnormally, by someone's pressing CTRL/C.

Usually you want your programs to terminate normally — that is, under the program's control. One of the best ways to terminate a program normally is to make sure that MINC executes a STOP or an END statement. These statements are defined in the following sections.

END Statements

An END statement when present must come as the very last statement in the program. When control passes to the END statement, the program terminates.

Programs will terminate by default by "falling out the bottom," that is, by looking for the statement after the last statement. However, the use of an END statement makes it clear that the programmer definitely wanted to end the program at a certain point — that no statements were forgotten at the end. Especially in a program where control must pass to a statement that terminates the program, END statements are clearer and better.

The STOP Statement

Like the END statement, the STOP statement terminates execution of a program. However, the STOP statement can be placed anywhere in a program. The STOP statement also prints a message giving the line number of the STOP statement that terminated execution.

CHAPTER 6
USING A REPETITIVE PROCESS

The power of a computer lies in its ability to perform tedious and repetitive processes quickly and to relieve people of such tasks. In fact, this capability can even result in changes to the nature of a solution. For example, when working by hand, you might approximate a solution to a problem; however, working with a computer, you might find you can do the entire set of computations in the same amount of time.

MINC will do repetitive processes if it is programmed to execute certain statements over and over again in a "loop". A *loop* is a sequence of statements that MINC repeats continuously until an end condition is met.

The sines and cosines problem executes some statements over and over and can calculate the sine and cosine of any number of angles. Again, the program is:

```
20 PRINT ,'SINE','COSINE'
30 PRINT 'DEGREES';
40 INPUT X
60 Z=X*PI/180
70 PRINT ,SIN(Z),COS(Z)
80 GO TO 30
```

MINC repeats statements 30, 40, 60, 70, and 80 until you type CTRL/C. What is actually happening is:

Statement	*What Happens*
20	print labels
30	prints prompt

Statement	*What Happens*
40	inputs angle in degrees
60	converts angle to radians
70	prints sine and cosine
80	returns control to line 30
30	prints prompt
40	inputs angle in degrees
60	.
70	.
80	.
30	

In this program, statements 30 through 80 form a loop.

LOOPS USING IF STATEMENTS AND GO TOs

A simple program that computes the squares and cubes of positive whole numbers follows:

```
10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 I=I+1
50 GO TO 30
RUNNH
```

I	I^2	I^3
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
.	.	.
.	.	.
.	.	.

Statements 30, 40, and 50 are the statements in the loop. This program does not terminate unless you press CTRL/C.

This program is in an *infinite loop*. Infinite loops never terminate by themselves. You must intervene and terminate the loop by pressing CTRL/C twice.

If you want a loop to terminate by itself, you can place a condition in the loop that terminates the loop when the condition is met. This condition is called the *end condition*. For example, this program can be rewritten to print the squares and cubes from 1 to 100 and stop after 100.

```
10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 IF I=100 GO TO 70
50 I=I+1
60 GO TO 30
70 END
RUNNH
```

I	I^2	I^3
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
.	.	.
.	.	.
.	.	.
99	9801	970299
100	10000	1.00000E+06

```
READY
```

This time the program terminates by itself after 100, 100^2, and 100^3 are printed.

Take care when programming the end condition of a loop. If you interchange statement 40 (the end test) and statement 50 (which increments the statement variable), then the test would no longer work as before. For example:

```
10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 I=I+1
50 IF I=100 GO TO 70
60 GO TO 30
70 END
```

The results of this loop are not as defined in the problem and are shown below. When I reaches 98, this is what happens:

Statement	*What Happens*
30	prints 98,98^2,98^3
40	increments I to 99
50	I (99) is not equal to 100 so go to 60
60	go to 30
30	prints 99,99^2,99^3
40	increments I to 100

Statement	*What Happens*
50	I (100) is equal to 100 so go to 70
70	end the program

The values 100, 100^2, and 100^3 are never printed because the end condition is not correct. In this case, statement 50 must be changed to:

```
50 IF I>100 GO TO 70
```

You should convince yourself (possibly by trying it) that this new line 50 allows 100, 100^2, and 100^3 to be printed.

To present another example of the many ways in which a loop can be written and still perform the same function, the following program is equivalent to the two previous versions of the squares and cubes problem.

```
10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 I=I+1
50 IF I<=100 GO TO 30
60 END
```

Although both the sines and cosines program and the squares and cubes program are loops, the logic of the programs is very different. The squares and cubes problem executes its loop exactly 100 times. The program counts the number of times the loop is executed. The number of times the sines and cosines loop is executed varies with each run and depends on when the user types CTRL/C.

Loops like that in the sines and cosines problem, which are terminated by a specific end condition, can most easily be programmed using IF and GO TO statements. However, loops like that in the squares and cubes problem, which are dependent on a count, can be done more easily with a FOR statement, which automatically counts the number of times the loop is executed.

FOR LOOPS AND THE FOR/NEXT STATEMENTS

Again, the squares and cubes program is listed below:

```
10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 I=I+1
50 IF I<=100 GO TO 30
60 END
```

Using the FOR statement you can rewrite the program like this:

```
10 PRINT 'I','I^2','I^3'
20 FOR I=1 TO 100
30 PRINT I, I^2, I^3
40 NEXT I
50 END
```

The results of the two versions are the same, but the second is one statement shorter.

The FOR and NEXT statements do the counting automatically. In the second example, the FOR statement:

```
20 FOR I=1 TO 100
```

sets up I as the variable that will count the loop, and I counts from 1 to 100. I is the *control variable* of the FOR loop. The FOR statement starts the top of the loop. The NEXT statement:

```
40 NEXT I
```

determines the bottom of the loop. The NEXT statement increments I by one and returns control to the top of the loop to start the loop over. When I gets incremented to greater than 100, control is passed to the statement following the NEXT (in this case, statement 50, the END statement).

There is only one difference between the way the two examples function, but the difference would not show in the output unless each program were changed slightly as follows.

```
10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 I=I+1
50 IF I<=100 GO TO 30
55 PRINT I
60 END
```

```
10 PRINT 'I','I^2','I^3'
20 FOR I=1 TO 100
30 PRINT I, I^2, I^3
40 NEXT I
45 PRINT I
50 END
```

In statement 55 of the first example above, MINC prints 101 as the value of I. In statement 45 of the second example above, MINC prints 100 as the value of I. The only difference between

```
20 FOR I = 5 TO 1 STEP -1
30 PRINT I
40 NEXT I
50 END
RUNNH

5
4
3
2
1

READY
```

The following example shows that any expression within the FOR statement can be fractional.

```
10 FOR J=0 TO 1.5 STEP .3
20 PRINT J
30 NEXT J
40 END
RUNNH

0
.3
.6
.9
1.2
1.5

READY
```

The next example shows that you can alter the control variable within the loop; however, you should take care to be sure that this is what you really want to do. (In fact, changing the control variable unintentionally is a major cause of program errors.)

```
10 FOR I = 1 TO 10
20 I = I + 1
30 PRINT I
40 NEXT I
50 END
RUNNH

2
4
6
8
10

READY
```

MINC never executes this last loop because I (10) is greater than the end value (1) right from the beginning.

```
10 FOR I=10 TO 1
20 PRINT I
30 NEXT I
40 END
RUNNH

READY
```

CHAPTER 7
ARRAYS AND NESTED LOOPS

An array is a group of conceptually similar variables, all with the same name. For example, suppose a professor has given a 20-question questionnaire to some students and asks you to process the results of this questionnaire using MINC. Your first idea for representing the questions might be to refer to each question on the questionnaire as Q1, Q2, Q3, and so forth. However, this method fails at Q10 because Q10 is not a valid BASIC variable name.

Instead, you can put the questions in an array named Q. You can then reference question 1 by the name Q(1). You can reference question 2 by the name Q(2). And, you can reference question 20 by the name Q(20).

In mathematical notation, these *array elements* are written as follows:

Q_0 Q_1 Q_2 Q_3 ... Q_{19} Q_{20}

The small number below the Q is called a *subscript*, and Q_2 is read "Q sub 2."

In MINC notation, the equivalent array elements are written as follows:

Q(0) Q(1) Q(2) Q(3) ... Q(19) Q(20)

In MINC, the number in parentheses is also called a subscript, and Q(0) is read "Q sub 0."

The questionnaire problem corresponds very well with the use of arrays. By using an array to represent the questionnaire in MINC, you give all the questions the same variable name, Q. Each individual question is distinguished by its unique subscript.

CREATING AN ARRAY

In order that MINC can reserve a workspace of sufficient size for your array of variables, you must define how many you will use. In BASIC you can use a dimension statement (DIM) to do this as follows.

```
10 DIM Q(20)
```

This statement tells MINC that you are defining an array named Q with a *dimension* of 20. The dimension determines the number of distinct elements in the array.

If you give an array a dimension of 20, MINC creates 21 distinct elements (elements 0 through 20). All arrays begin with element 0.

The general form of the DIM statement is:

DIM array-list

The argument array-list is the list of arrays to be dimensioned in this statement. An array in the list is written as follows:

array-name(dimension)

The array-name must follow the rules for numeric variable names if the elements in the array are numeric values. The array-name must follow the rules for string variable names if the elements in the array are going to be strings.

The dimension represents the maximum subscript that you can use in the array. The dimension is the number of elements in the array minus one (because of element 0). In the questionnaire example, the dimension is 20 because there are 20 questions in the questionnaire.

The maximum number of elements that you can define in an array depends on the size of the program.

Because the array and the program must both fit in the workspace, a short program leaves more room for a large array, and a large program requires a shorter array.

Remember that with a dimension of 20, there are really 21 elements. For convenience, element 0 will be ignored for now. In later sections, this chapter discusses how to adjust the program to use element 0.

WHY USE ARRAYS?

You should use an array for the questionnaire problem because it is very inefficient to name the questions Q1, Q2, and so forth. Even if MINC allowed variable names like Q20 (which is does not), the programming to input all 20 questions would be a problem. You would have to type the following INPUT statement.

```
10 INPUT Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10
11 INPUT Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,Q19,Q20
```

If you wanted to make your program more practical by prompting each question, you would have to type the following 20 lines just to input the answers to one questionnaire:

```
10 PRINT '1';\ INPUT Q1
20 PRINT '2';\ INPUT Q2
30 PRINT '3'\ INPUT Q3
.
.
.
```

By using an array, you can take advantage of FOR loops. Even prompting each question becomes easy. The part of the program to input the answers to one questionnaire is:

```
10 DIM Q(20)
40 FOR I=1 TO 20
50 PRINT I; \ INPUT Q(I)
60 NEXT I
```

Instead of using 20 lines of code, this method uses four. The way this loop works is:

Statement	*What Happens*
40	starts I at 1
50	prints 1, inputs Q(1) (because I=1)
60	increments I to 2
50	prints 2, inputs Q(2) (because I=2)
60	increments I to 3
.	.
.	.
.	.
60	increments I to 20

Statement	*What Happens*
50	prints 20, inputs Q(20) (because I = 20)
60	increments I to 21, exits loop, decrements I to 20

Because you can use arrays so easily in FOR loops, arrays make programming much easier for problems like this one. The computer does the repetition instead of the programmer.

MINC allows up to two subscripts for an array. An array with one subscript is called a *one-dimensional array*. An array with two subscripts is called a *two-dimensional array*. These two kinds of arrays are discussed in the following sections.

ONE-DIMENSIONAL ARRAYS

So far the method used for storing the results of the questionnaire has been to store the results in a *one-dimensional array*. A one-dimensional array can be pictured as follows. This particular one-dimensional array is named Q.

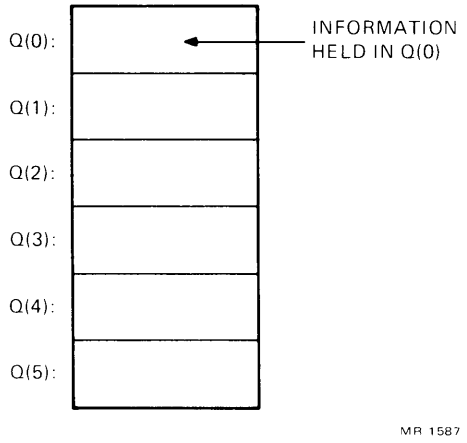


Figure 7. One-Dimensional Array

This one-dimensional array has six elements (0 through 5).

From the examples in the previous sections, you now know how to get the data from one questionnaire into a one-dimensional array in MINC. Now you have to devise a scheme to process the data.

Suppose the questionnaire has 20 questions, each with three possible responses: agree, disagree, and don't care.

For the time being, all the professor wants is to keep a tally for each question of the number of agrees, disagrees, and don't cares

received. One method is to type in the following input for each question.

Enter a 1 for agree.
Enter a 2 for disagree.
Enter a 3 for don't care.

A program to compute the results of these responses can use an ON/GO TO statement. Besides needing an array for the input of each questionnaire, you need 3 more arrays — one to tally agrees, one to tally disagrees, and one to tally don't cares. Statement 10 becomes:

```
10 DIM Q(20), A(20), D(20), C(20)
```

For example, A(1) tallies the number of people who agreed with question 1. D(15) tallies the number of people who disagreed with question 15. C(12) tallies the number of people who don't care about question 12, and so forth. (Remember that element 0 of each array is being ignored for the time being.)

The program to read and process the questionnaire now becomes:

```
10 DIM Q(20),A(20),D(20),C(20)
20 REM - Q represents the original questionnaire
30 REM - A represents the number of people who agreed
35 REM - with each question
40 REM - D represents the number of people who disagreed
45 REM - with each question
50 REM - C represents the number of people who don't
55 REM - care about each question
60 REM - Input the questionnaire
70 FOR I=1 TO 20
80 PRINT I; \ INPUT Q(I)
90 NEXT I
100 REM - Process questionnaire
110 FOR I=1 TO 20
120 ON Q(I) GOTO 150, 160, 170
150 A(I)=A(I)+1 \ GOTO 200
160 D(I)=D(I)+1 \ GOTO 200
170 C(I)=C(I)+1
200 NEXT I
```

In line 120, if Q(I), the response to question number I, is 1, the agree counter is incremented. If Q(I) is 2, the disagree counter is incremented. If Q(I) is 3, the don't care counter is incremented. The ON/GO TO statement ensures that only one counter is incremented for each question.

Because MINC sets all variables to 0 when you type the RUN command, the counters are all started at 0. You do not have to give the counter arrays an initial value of zero.

You can shorten this program segment to input and process a questionnaire. Q is an unnecessary array. You do not need to read in all 20 questions at once when you can process one question at a time. By eliminating the Q array, you can save 19 variables in the workspace. You need only one variable, rather than a whole array of 20 variables.

Without changing the basic algorithm or the approach, you can change the program to reduce the number of variables as follows.

```
10 DIM A(20),D(20),C(20)
30 REM - A represents the number of people who agreed
35 REM - with each question
40 REM - D represents the number of people who disagreed
45 REM - with each question
50 REM - C represents the number of people who don't
55 REM - care about each question
60 REM - Input and process one questionnaire
70 FOR I=1 TO 20
75 REM - R represents one response to one question
80 PRINT I; \ INPUT R
100 REM - Process question
120 ON R GOTO 150, 160, 170
150 A(I)=A(I)+1 \ GOTO 200
160 D(I)=D(I)+1 \ GOTO 200
170 C(I)=C(I)+1
200 NEXT I
```

There is another method that uses two-dimensional arrays for inputting and processing a questionnaire. This method is discussed in the next section.

TWO-DIMENSIONAL ARRAYS

The previous algorithm for processing questionnaires uses three one-dimensional arrays. Again, a one-dimensional array uses one subscript and can be pictured as shown in Figure 8.

A *two-dimensional array* can be thought of as a matrix and requires two subscripts. Two-dimensional arrays can be pictured as shown in Figure 9.

You can use one two-dimensional array to replace the three one-dimensional arrays in the questionnaire problem. Array R,

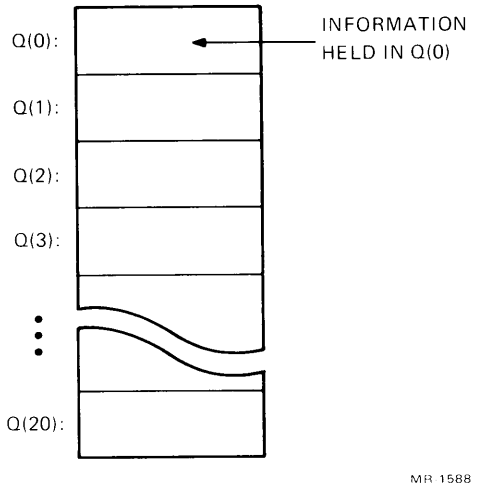


Figure 8. One-Dimensional Array

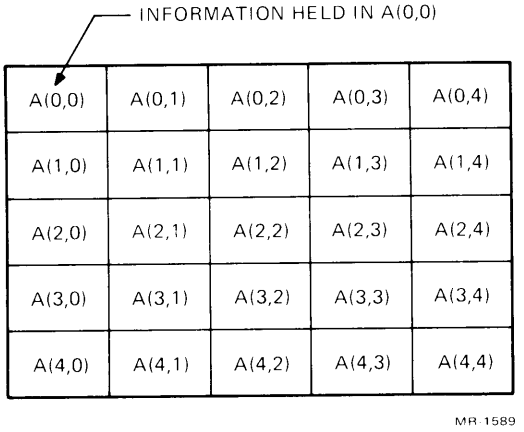


Figure 9. Two-Dimensional Array

which stands for response, will be dimensioned as follows:

```
10 DIM R(20,3)
```

The first subscript (ranging from 0 through 20) represents the question being answered. The second subscript (1 through 3) represents the response. Therefore, the value of R(2,1) represents the number of people who agreed with question 2. (Remember that all elements with a subscript of 0 are being ignored.)

This method has no fewer variables because three one-dimensional arrays that are 20 elements long have as many variables as one two-dimensional array with 20 x 3 variables. However, the program becomes much shorter. (In fact, this

method is wasting a few more variables by ignoring Q(0,0) through Q(20,0). However, using these elements is discussed in later sections.)

Use of a two-dimensional array simplifies the loop to input and process questionnaires to the following four statements:

```
70 FOR I=1 TO 20
80 PRINT I; \ INPUT Q
90 R(I,Q)=R(I,Q)+1
100 NEXT I
```

The first subscript represents the number of the question; the second subscript represents the code for the answer. In line 90, if I is equal to 12 and Q is equal to 2, then R(12,2) is incremented.

NESTED LOOPS

A very simple program is now developed for processing one questionnaire. However, the program must be expanded to process many questionnaires. The program needs a loop to process more than one questionnaire; but, you cannot use a FOR loop because you do not know how many questionnaires there are to be processed. Instead, you must create a loop that will work for any number of questionnaires. The following program segment presents a loop for processing many questionnaires.

```
20 PRINT 'Input another questionnaire'; \ INPUT A$
30 IF A$='YES' GO TO 70
40 IF A$= 'NO' GO TO 350
50 GO TO 20
60 REM - Input and process one questionnaire
70 FOR I=1 TO 20
.
.
.
110 GO TO 20
350 END
```

Many people do not test specifically for a "NO" or invalid response (that is, something other than yes or no) as in lines 40 and 50. Their test would be:

```
50 IF A$<>'YES' GOTO 350
```

However, if you are inputting questionnaire number 199 and you mistype yes for questionnaire 200, you do not want to have to start typing all over again. Thus, it is a good idea to test for the proper input.

The program segment for inputting and processing many questionnaires is shown as follows:

```
10 DIM R(20,3)
20 PRINT 'Input another questionnaire'; \ INPUT A$
30 IF A$ = 'YES' GO TO 70
40 IF A$ = 'NO' GO TO 350
50 GO TO 20
60 REM - Input and process one questionnaire
70 FOR I = 1 TO 20
80 PRINT I; \ INPUT Q
90 R(I,Q) = R(I,Q) + 1
100 NEXT I
110 GO TO 20
350 END
```

This program has one loop (the FOR loop) within another loop (the IF/GO TO loop). This process of putting one loop inside another is called *nesting* loops. You can also nest FOR loops if you need to.

These nested loops work exactly the way this problem was solved. The *inner loop*, which includes statements 70 to 100, is processed first. That is, the program processes all 20 questions of one questionnaire — by repeating statements 70 through 100 twenty times. Once the entire questionnaire is processed, control passes to statement 110, and the outer loop is processed. If there is another questionnaire, control passes back to statement 70 which will again start the inner loop, and the whole process described in this paragraph is repeated. If there is not another questionnaire, the program will stop.

This program only reads and processes the questionnaire. To make the program of any value to the professor, you must produce a report of the results.

The following program produces a report that lists the results.

```
10 DIM R(20,3)
15 REM - Input next questionnaire
20 PRINT 'Input another questionnaire'; \ INPUT A$
30 IF A$ = 'YES' GO TO 70
40 IF A$ = 'NO' GO TO 310
50 GO TO 20
60 REM - Input and process one questionnaire
70 FOR I = 1 TO 20
75 REM - Q represents the response to one question
80 PRINT I; \ INPUT Q
85 REM - Process question
90 R(I,Q) = R(I,Q) + 1
100 NEXT I
```

```
110 GO TO 20
300 REM - Print results of all questionnaires
310 PRINT ,,'AGREE', 'DISAGREE', 'DON'T CARE'
320 FOR I=1 TO 20
330 PRINT 'Question';I,,R(I,1), R(I,2), R(I,3)
340 NEXT I
350 END
```

Below is a partial sample run of this program. This sample does not show the entire input process or the entire output.

```
RUN

QUEST                 28-FEB-80              10:27:00

Input another questionnaire? YES
1?  1
2?  3
3?  1
4?  2
.
.
.
20?  3
Input another questionnaire? NO
```

	AGREE	DISAGREE	DON'T CARE
Question 1	23	10	7
Question 2	5	29	6
Question 3	18	4	18
Question 4	2	38	0
.	.	.	.
.	.	.	.
.	.	.	.

The above example of the QUEST program now does the entire job of taking questionnaires as input, processing the results, and printing the results. However, this program does not process the questionnaires well. You should be aware of the following problems:

- The program expects an upper case YES as the answer to "Input another questionnaire" and does not recognize a lower case yes.

- The program expects the person typing in the questionnaire responses to be a perfect typist. The program does not validate that the current response is between 1 and 3. In the above example, if the response is other

than 1, 2, or 3, MINC will have a problem as it tries to update R(I,Q) in line 90. The value of Q will not be within the dimensions of the array R. MINC prints the following message:

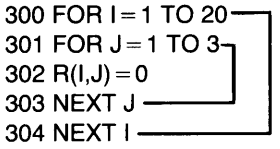
```
?MINC-F-Array subscript is negative or too large at line 90
```

Techniques for resolving these and similar problems are discussed in Chapters 8 and 11.

Nested FOR Loops

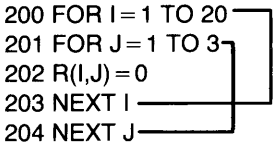
Suppose for some reason you decided that you want to reset your two-dimensional array R to zero. That is, you want to set each element to zero later in the program without having to run the program again. You can accomplish this by several means, some easier than others.

The easiest way to access each element of a two-dimensional array is to use nested FOR loops (as shown in the following example).



Nested FOR loops must be completely nested. That is, the inner FOR loop must be entirely within the outer FOR loop as shown by the lines drawn in on the previous example.

Here is an example of an invalid pair of nested FOR loops.

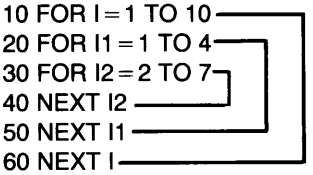


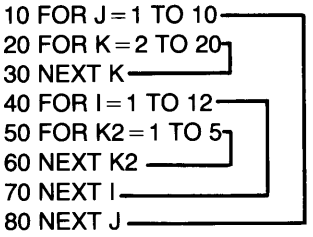
At statement 204, what is the value of I? MINC gets confused at this line, tries to execute the loop, cannot accept this loop, and prints the following message.

```
?MINC-F-No corresponding FOR statement for NEXT at line 203
```

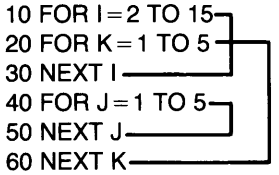
Here are some examples of acceptably and unacceptably nested FOR loops.

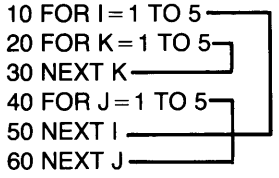
Acceptable FOR loops





Unacceptable FOR loops





USING ARRAY ELEMENT 0

The previous examples in this chapter ignored element 0 of each array. The next two sections discuss how you can modify your algorithms to use element 0 for one- and two-dimensional arrays.

One-Dimensional Arrays

Remember that when you dimension a one-dimensional array, it actually has one more element than the value of the dimension implies. For example, the following array has six elements, even though its dimension is 5.

```
10 DIM A(5)
```

In the professor's problem, Q(0), A(0), D(0), and C(0) were ignored, and the FOR loops started at 1. Because the questionnaires started with question 1, not question 0, the programming was made easier by ignoring element 0 in each of the arrays.

A FOR loop to access every element of the six-element array follows:

```
10 FOR I=0 TO 5
20 A(I)=1
30 NEXT I
```

In this FOR loop, every element is given an initial value of 1.

Two-Dimensional Arrays

Both dimensions of a two-dimensional array start with zero. For example, the following array:

```
10 DIM A(4,4)
```

actually looks like Figure 9 (Page 99).

The two-dimensional solution to the professor's problem ignored row zero and column zero of the arrays and started the FOR loops at 1. Because the array was dimensioned 20 by 3, the program wasted 20+3+1 array elements, or 24 variables. You really don't want to waste space (that is, declare unused variables) if possible. You can save the space by changing the program:

```
10 DIM R(19,2)
.
.
.
70 FOR I=0 TO 19
80 PRINT I+1;\ INPUT Q
90 R(I,Q-1)=R(I,Q-1)+1
100 NEXT I
.
.
.
```

The array R now has 60 elements. The FOR loop still executes 20 times. Line 80 still prints the right question number; 0+1 is 1 for the first question. Q-1 converts the agree, disagree, and don't care codes to 0, 1, and 2. You could change the input to 0, 1, and 2, but the programmed conversion is just as simple. Your choice of algorithm depends on how you want to structure your problem. In this case, questionnaires rarely start with question 0, so the program makes the conversion from question 1 to element 0.

This program makes better use of the workspace by not wasting variables. Better yet, the program change is invisible to anyone

using the program. That is, the person inputting the questionnaires and printing the results for the professor does not have to learn a new code for the input or have to do anything different, even though the program has changed.

SUBSCRIPTED VARIABLES

Arrays are a group of variables all given the same name. The individual elements are distinguished by the one or two unique subscripts. Each element then is called a *subscripted variable.*

Subscripts

The subscripts of a variable determine the unique array element being referenced. Picture a one-dimensional array that looks like Figure 10.

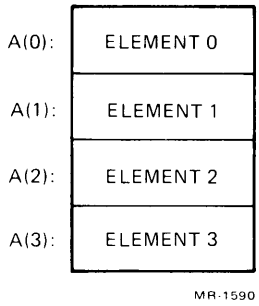


Figure 10. One-Dimensional Array

Array A has four unique elements. The first element is specifically referenced by the name A(0). The fourth element is specifically referenced by the name A(3). As you can see in Figure 10, each array element is a discrete unit, and referencing A(3.7) does not really make any sense. Therefore, before MINC references an array element, it truncates any fractional part of all subscripts to make them whole numbers. Thus, in the following statements, reference A(3), the value of I remains 3.7, but in statement 20, A(3), not A(3.7), gets the value 7.

```
10 I=3.7
20 A(I)=7
```

A subscript can be any numeric expression. For example, A(I+3*B^2) is a perfectly valid reference to a subscripted variable (as long as I+3*B^2 is within the dimension of the array). B(Q+7,I*2.65+9) is a valid reference to a subscripted variable for the two-dimensional array B. If either or both subscripts are not whole numbers, MINC truncates each subscript to a whole number before accessing the array element.

Subscripts can never be string literals or string variables. Subscripts must always be numeric.

Subscripts must remain within the *bounds* or *dimensions* of the array. For example, if you dimension the following array:

```
10 DIM C(8)
```

MINC will not let you reference C(-5) or C(9), nor will it let you use any subscript that is not between 0 and 8. Should you try to reference elements that are out of bounds, MINC will issue the following error message and terminate your program.

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

When using a two-dimensional array, you must take care that both dimensions are within the bounds set in the DIM statement. For example, if you dimension the following array:

```
10 DIM C1(3,5)
```

the first subscript must be between 0 and 3, and the second subscript must be between 0 and 5. C1(4,3) is an invalid reference because the first subscript is too large. C1(1,6) is an invalid reference because the second subscript is too large. C1(-1,8) is an invalid reference because both subscripts are out of bounds.

If the subscripts of a two-dimensional array are out of bounds, MINC will terminate the program with the following message.

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

Examples

Example 1

This example counts the number of times each number from 0 to 10 is entered by the person running the program (you, if you try this example).

```
NEW COUNT
10 DIM A(10)
20 FOR I=1 TO 5
30 PRINT 'Input a number from 0 to 10'; \ INPUT N
40 A(N)=A(N)+1
50 NEXT I
60 PRINT
100 FOR I=0 TO 10
110 PRINT 'The number';I;'appeared';A(I);'time(s).'
120 NEXT I
130 END
RUN
```

```
COUNT                01-MAR-80              15:59:55

Input a number from 0 to 10? 5
Input a number from 0 to 10? 3
Input a number from 0 to 10? 1
Input a number from 0 to 10? 0
Input a number from 0 to 10? 5

The number 0 appeared 1 time(s).
The number 1 appeared 1 time(s).
The number 2 appeared 0 time(s).
The number 3 appeared 1 time(s).
The number 4 appeared 0 time(s).
The number 5 appeared 2 time(s).
The number 6 appeared 0 time(s).
The number 7 appeared 0 time(s).
The number 8 appeared 0 time(s).
The number 9 appeared 0 time(s).
The number 10 appeared 0 time(s).

READY
```

There are no problems with this example. However, when MINC displays READY, if you type RUN and enter the numbers 1 and 11, MINC will halt the program. In line 40, MINC tries to increment A(11), cannot find A(11), and terminates the program.

Example 2

The following short program written by a history professor tallies votes for the Moderate and Reform parties in all elections between 1900 and 1915. As input, it takes the year and the party — M for Moderate and R for Reform.

```
10 DIM E(15,1)
20 PRINT 'Enter year,party'; \ INPUT Y,P$
30 REM - validate party
35 IF P$<>'r' THEN IF P$<>'m' GO TO 20
40 IF P$='m' THEN P=0
50 IF P$='r' THEN P=1
60 REM - validate year
70 IF Y<1900 GO TO 20
80 IF Y>1915 GO TO 20
85 REM - update tally
90 E(Y-1900,P)=E(Y-1900,P)+1
100 REM - next input
110 PRINT 'more data'; \ INPUT R$
120 IF R$='yes' GO TO 20
130 FOR I=0 TO 15
140 PRINT 'year';I+1900;'moderate votes';E(I,0);'reform votes';E(I,1)
150 NEXT I
160 END
```

This program checks the input data to ensure that each piece of data is correct before the program tries to access an array element. If the data are not correct, the program reissues the prompt and waits for valid data.

Example 3

In the following example, the subscript goes outside the bounds of the array, but the error is not obvious.

```
10 DIM A(10)
20 FOR I=0 TO 10 STEP 2
30 PRINT A(I),A(I+1)
40 NEXT I
50 END
```

In line 30, when I=10, I+1=11. A(11) is out of the bounds of the array A.

Subscripted Variables

Subscripted variables are array elements, or variables, that are referred to by the use of subscripts. Although the subscripts themselves must always be numeric, the variables can hold either numeric or string data (depending on the type of array).

Subscripted string variables can be used anywhere that string variables can be used. Subscripted numeric variables can be used anywhere that numeric variables can be used except as the control variable for a FOR loop.

The following example uses a string array. This program accepts as input students' names and their three test scores. The program prints out the students' names, test scores, and final grade based on the test scores.

```
10 DIM S$(25),G1(25),G2(25),G3(25),G$(25)
20 PRINT 'Input student name and three test scores';
25 INPUT S$(I),G1(I),G2(I),G3(I)
30 F=G1(I)+G2(I)+G3(I)
40 REM - compute student's letter grade based on average
45 REM - of the three test scores
50 F=F/3
55 IF F<60 THEN G$(I)='F'
60 IF F>=60 THEN IF F<70 THEN G$(I)='D'
70 IF F>=70 THEN IF F<80 THEN G$(I)='C'
80 IF F>=80 THEN IF F<90 THEN G$(I)='B'
90 IF F>=90 THEN G$(I)='A'
100 PRINT 'Another student'; \ INPUT R$
110 IF R$='no' GO TO 150
120 IF R$<>'yes' GO TO 100
```

```
125 REM - count the next student
130 I=I+1
140 GO TO 20
150 PRINT 'STUDENT NAME','GRADE 1','GRADE 2','GRADE 3','FINAL GRADE'
160 FOR J=0 TO I
170 PRINT S$(J),G1(J),G2(J),G3(J),G$(J)
180 NEXT J
190 END
RUNNH

Input student name and three test scores? Andrea,90,93,95
Another student? yes
Input student name and three test scores? Jason,85,75,80
Another student? no

STUDENT NAME    GRADE 1    GRADE 2    GRADE 3    FINAL GRADE
Andrea              90         93         95              A
Jason               85         75         80              B

READY
```

HOW ARRAYS ARE STORED IN THE WORKSPACE

It is not necessary for you to read the following discussion of arrays unless you intend to use some of the specialized MINC program features (see Books 4 and 6).

When you dimension an array with a DIM statement, MINC sets up an area in the workspace for the array.

One-dimensional arrays are stored as they have been previously pictured. Thus, the array dimensioned in the following statement:

```
10 DIM A(5)
```

is stored in the workspace as shown in Figure 11.

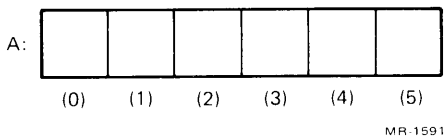


Figure 11. One-Dimensional Array

Although you can think of two-dimensional arrays as matrices, they are stored linearly in the workspace. Thus, the array dimensioned in the following statement:

```
10 DIM B(2,3)
```

can be pictured as in Figure 12.

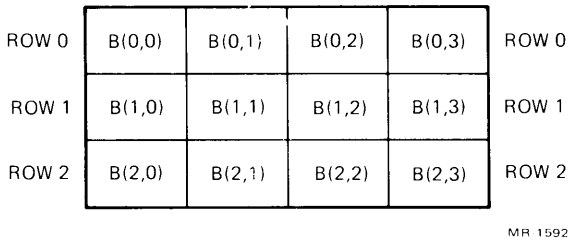


Figure 12. Two-Dimensional Array

However, this array is stored in the workspace as shown in Figure 13.

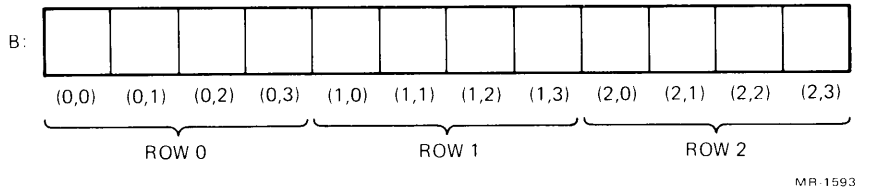


Figure 13. Two-Dimensional Array Stored in the Workspace

MINC stores a two-dimensional array in *row-major order*. That is, MINC stores all of row zero, then all of row one, and so forth. In this order, the rightmost subscript varies the fastest as you look down the linear list of the two-dimensional array.

When working with a two-dimensional array in a BASIC program, you must always use two subscripts when referencing one of the array elements. Thus, the way the array is stored in the workspace is usually of little importance to you.

However, some aspects of the lab module routines discussed in Book 6 are easier to use if you know how two-dimensional arrays are stored in the workspace.

CHAPTER 8
DATA TYPES AND FUNCTIONS

Chapter 2 discussed numeric variables and numeric functions (the trigonometric functions and some arithmetic functions). This chapter treats data types and numeric variables in more detail and discusses numeric and string functions.

DATA TYPES

So far, this manual has dealt with two data types: numeric and string. String variables and literals have been discussed completely; however, numeric variables and literals have been simplified until now.

There are two kinds of numeric variables and literals: real and integer. These numeric data types are discussed in the following sections.

Real Variables and Literals

Those variables and literals that have previously been described as numeric are *real variables* and *literals*. Real numbers can have a fractional part or they can be whole numbers. For example, 7 and 7.3 are both real literals. The limits on real literals are 10^{-38} to 10^{38}.

Real variables are named by a letter, or a letter followed by a digit. Thus, A and A1 are two real variable names. A program can have at most 286 real variables (A-Z, and A0-A9 through Z0-Z9).

Integer Variables and Literals

Integer variables and *literals* can have only whole number values. If you try to give an integer variable a fractional value, MINC truncates the value to the whole number portion.

To distinguish an integer literal from a real literal, you must type a percent sign (%) after the literal. For example, the following literal is a real literal.

7

The integer literal 7 is denoted as follows:

7%

The limits on integer literals are -32,768 to 32,767.

Integer variables can take on only whole number values. Like integer literals, integer variables never have a fractional part. Integer variables have the same range of values as integer literals, -32,768 to +32,767.

MINC integer variable names are represented with a letter followed by a percent sign, or with a letter followed by a digit followed by a percent sign. For example, A% and A2% are two distinct integer variable names.

MINC stores real numbers and integers differently. A real number takes twice as much room in the workspace as an integer, but a real number has more precision (the fractional part) than an integer and has a different range.

You must be careful when you perform arithmetic operations using integer variables because fractional results of any operation are truncated.

Operations involving only integer operands are called *integer arithmetic*. Operations involving a combination of integer and real operands are called *mixed mode* arithmetic.

NOTE

The term *whole number* is used to denote a number with no fractional part. A whole number can be stored in a real variable or can be a real literal. The term *whole number* does not denote a data type. In contrast, the term *integer* refers to the *data type*. An integer literal must be followed by a percent sign. Integers must be whole numbers, but whole numbers are not necessarily integers.

Integer Arithmetic

MINC handles operations that use only integer operands quite differently than it does those involving real numbers. For example, because there are no decimal fractions allowed with integers, the following division results in a value of 0.

```
PRINT 1%/7%
  0
```

The above example, and the following examples in this section, are not trying to show good use of MINC. That is, you would probably never print the value 1%/7%. However, while using MINC you might want to do a division using integer variables, or perform an operation using a mixture of integer variables and real literals or variables. The following examples show you how to do these operations. To make the examples clear, this section will use literals rather than variables, even though you would never do these specific types of operations with literals.

Again, the limits on integer values are -32768 to +32767. If you exceed these limits, MINC will print an error message. For example:

```
PRINT 3% * 30000%
?MINC-W-Value of integer expression not in range -32768 to +32767
  0
```

MINC warns you that it cannot handle an integer as large as 90,000 and substitutes 0 for the result of the calculation.

If you assign a real value to an integer variable, MINC truncates the value; that is, it cuts off the fractional portion. For example:

```
A% = 7.999
PRINT A%
  7
```

Mixed Mode Arithmetic

You can perform calculations with a mixture of integer and real literals and variables. However, unless you understand how MINC performs the *mixed mode arithmetic*, you can obtain very unexpected results.

Real numbers take precedence over integers. That is, a combination of real and integer operands results in a real result. For example, in the first division below, the real operand (15) produces a real result (.666667). In the second division, both operands are integer, producing an integer result.

```
PRINT 10%/15
  .666667
```

However:

```
PRINT 10%/15%
  0
```

The order in which MINC does the calculations can affect the result. In the following example, MINC performs the calculation from left to right because division and multiplication are of equal priority. The integer division is done before the real multiplication, resulting in a value of 0.

```
PRINT 10%/15%*20
  0
```

This same calculation follows, but in a different order. In this second example, the multiplication is done first, resulting in a real interim result, which causes the division to produce a real result.

```
PRINT 20*10%/15%
  13.3333
```

Parentheses override operator precedence in mixed mode arithmetic. For example:

```
PRINT 20*(10%/15%)
  0
```

When MINC assigns a real number to an integer variable, it truncates the value before making the assignment. For example:

```
A% = 15/10

PRINT A%
  1
```

The following chart sums up the results of mixed mode operations:

operand 1	*operand 2*	*result*
real	real	real
real	integer	real
integer	real	real
integer	integer	integer

ARITHMETIC AND TRIGONOMETRIC FUNCTIONS

In Chapter 2, the following arithmetic and trigonometric functions were discussed.

Arithmetic functions

SQR	the square root function
EXP	the exponential function
LOG, LOG10	the logarithmic functions

Trigonometric functions

PI	the function that computes the value of π
SIN	the sine function
COS	the cosine function
ATN	the arc tangent function

You can use the arithmetic and trigonometric functions in a program as easily as in the immediate mode because the argument of a function can include variables as well as constants.

For example, the following program computes the nonimaginary roots of the following quadratic equation.

$$0 = ax^2 + bx + c$$

This program uses the quadratic formula given below.

$$\frac{-b \pm \sqrt{(b^2-4ac)}}{2a}$$

```
10 PRINT 'Input a, b, and c';\ INPUT A,B,C
20 D=B^2-4*A*C
25 IF D>=0 GO TO 40
30 PRINT 'Roots are imaginary'
35 GO TO 70
40 R1=(-B+SQR(D))/2*A
50 R2=(-B-SQR(D))/2*A
60 PRINT 'The roots are';R1;'and';R2
70 END
```

The general form of an arithmetic or trigonometric function is:

function name (argument)

The argument can be any arithmetic expression. The remaining arithmetic functions available with MINC are:

INT	calculates the integer portion of its argument
ABS	calculates the absolute value of its argument
RND	produces a random number
SGN	computes the sign of its argument

These arithmetic functions are described in the following four sections.

Integer Function

The integer function (INT) takes on the value of the greatest integer less than the value of the argument. If the argument is not a whole number, the function truncates the fractional part.

The form of the integer function is:

INT(expression)

For example:

```
PRINT INT(7)
  7

PRINT INT (-7.35)
 -8

PRINT INT (7.3E-9)
  0
```

Absolute Value Function

The absolute value function (ABS) takes on the absolute value of the argument of the function. The form of the absolute value function is:

ABS(expression)

For example:

```
PRINT ABS (-3)
  3

PRINT ABS (-16.25)
  16.25
```

Random Number Function and RANDOMIZE Statement

The random number function (RND) generates a pseudo-random number between 0 and 1 each time it is invoked. The term *pseudo-random number* is used because MINC computes the random numbers according to a formula that ultimately repeats its sequence of numbers. However, this sequence of numbers is

so large that the numbers can be considered random.

For example:

```
PRINT RND
  .0407319
```

Following every RUN or SCR command, the RND function will begin at the same point in the set of random numbers. This feature helps you find potential errors in your program by producing the same conditions for each run.

To cause MINC to produce a set of random numbers with a new starting point for each run, use the RANDOMIZE statement in the immediate mode or as a statement in your program. For example:

```
READY
SCR

READY
PRINT RND
  .0407319

READY
SCR

READY
PRINT RND
  .0407319

READY
SCR

READY
RANDOMIZE

READY
PRINT RND
  .86727

READY
SCR

READY
RANDOMIZE

READY
PRINT RND
  .169211

READY
```

The RANDOMIZE statement is a BASIC statement and not a function. It does not calculate a value. Its only purpose is to cause MINC to start the random numbers at various points in the sequence.

Computing the Sign of an Expression

The sign function (SGN) takes on the value +1 if the argument is positive, a value of -1 if the argument is negative, or a value of 0 is the argument is 0.

The form of the SGN function is:

SGN (arithmetic expression)

For example, in the following assignment statement, X is assigned the value -1.

```
X=SGN(-3)

PRINT X
 -1
```

Note that the following mathematical relationship is true in MINC.

```
X=SGN(X)*ABS(X)
```

STRING FUNCTIONS

When you use a string variable or string literal, MINC stores the characters in the string in the workspace. MINC stores characters in a code called the ASCII code. This code represents each character with a numeric code.

For example, the internal representation of the character '2' is very different from the internal representation of the number 2. For example, the decimal value of the internal representation of the number 2 is 2. The decimal value of the ASCII code for the character '2' is 50. The ASCII code for 'A' is 65 and the ASCII code for 'a' is 97. The ASCII codes are all in the range 0 to 255. For a full list of the characters and their ASCII codes, see the table in Appendix A.

MINC provides the string functions that allow you to examine and modify strings of ASCII characters and perform certain string-to-numeric conversions.

String functions that produce a string value have a dollar sign ($) at the end of their name. String functions that produce real or integer numbers do not have a dollar sign.

Clock and Calendar Functions

When you start your MINC, you have to type the date and time. MINC has functions that let you determine the current values MINC has for the date and time.

The CLK$ function produces the time, measured on a 24-hour clock. The DAT$ function produces the date. For example:

```
Please enter
Today's date: 26-APR-80
Current time: 11:28

READY
PRINT CLK$
11:28:37

READY
PRINT CLK$
11:28:43

READY
PRINT DAT$
26-APR-80

READY
```

The CLK$ and DAT$ functions have no arguments.

The TIME and DATE Commands

If, when using MINC, you want to change the time or date that MINC has stored, you can use the TIME and DATE commands. The TIME command sets the MINC system clock, and the DATE command sets the system date. The form of the TIME command is:

TIME hh:mm:ss

where hh stands for hours, mm stands for minutes, and ss stands for seconds. The minutes and seconds as well as the trailing colons are optional. The MINC system clock is a 24-hour clock.

The form of the DATE command is:

DATE dd-mmm-yy

where dd stands for the day of the month, mmm stands for the first three letters of the month, and yy stands for the last two digits of the year.

For example:

```
TIME 17:15

READY
DATE 1-FEB-80

READY
PRINT CLK$
17:15:35

READY
PRINT DAT$
1-FEB-80

READY
```

String Manipulation Functions

The MINC string manipulation functions allow you to do the following:

- Determine the length of a string (LEN).
- Trim off trailing blanks from a string (TRM$).
- Search for the position of a set of characters within a string (POS).
- Copy a segment from a string (SEG$).

Remember that the string operation is concatenation (& or +). The string relational operators test for equality of strings (=, <>) and alphabetic precedence (<, >, <=, >=).

Finding the Length of a String

You can use the LEN function to find the length, or number, of characters in a string. The LEN function produces a whole number value equal to the length of the string you specify. The form of the LEN function is:

LEN(string)

The following example prints the length of a string containing all the letters in the alphabet.

```
10 A$ = 'abcdefghijklmnopqrstuvwxyz'
20 PRINT LEN(A$)
RUNNH

 26

READY
```

Trimming Trailing Blanks Off a String

The TRM$ function produces the argument string with any trailing blanks removed. The form of the TRM$ function is:

TRM$(string)

The following example concatenates and prints two strings both before and after it trims trailing blanks.

```
10 A$ = 'abcd'
20 B$ = 'efg'
30 PRINT 'before trimming:';A$&B$
40 PRINT 'after trimming:';TRM$(A$)&B$
RUNNH

before trimming: abcd efg
after trimming: abcdefg

READY
```

Finding the Position of a Substring

You can use the POS function to see if a group of characters (*substring*) occurs within a larger string. The form of the POS function is:

POS(search-string,substring,numeric-expression)

where:

search-string	is the string being searched.
substring	is the substring the POS function is searching for.
numeric-expression	is the position in the search-string at which MINC starts the search.

The POS function searches for and produces a whole number equal to the first occurrence of the substring in the search-string. POS begins the search with the character position specified by numeric-expression. If POS finds the specified substring, it produces the character position of the first character of the substring relative to the beginning of the entire string. If POS does not find the specified substring, it produces a 0.

The following example translates each name of a month to its numeric equivalent (for example, APR to 4). In line 140 the POS function returns the position of the input string M$ in the string containing the first three letters of each month, T$.

If the program finds the month you specify, it prints the number of the month. If it does not find the month, it requests you to try again.

```
10 T$ = 'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
100 PRINT 'type the first 3 letters of a month'; \ INPUT M$
120 IF M$ = 'END' GO TO 32767
130 IF LEN(M$)<>3 GO TO 200
140 M = (POS(T$,M$,1) + 2)/3 \ REM - find the number of the month
150 IF M<>INT(M) GO TO 200
160 PRINT M$;'is month number';M
170 GO TO 100
200 PRINT 'invalid entry - - - try again' \ GO TO 100
32767 END
RUNNH

type the first 3 letters of a month? NOV
NOV is month number 11
type the first 3 letters of a month? MAY
MAY is month number 5
type the first 3 letters of a month? JUD
invalid entry - - - try again
type the first 3 letters of a month? END

READY
```

There are certain boundary conditions that are dependent on the values of the strings and the expression. For the details of these boundary conditions, see Book 3.

Copying Segments from a String

You can use the SEG$ function to produce a segment (*substring*) of a string. The SEG$ function produces a substring consisting of the characters in the string you specify between specified character positions. The original string is unchanged. For example, the following call to SEG$ prints characters 3 through 5 of string 'ABCDEF'.

```
10 PRINT SEG$('ABCDEF',3,5)
RUNNH

CDE

READY
```

The form of the SEG$ function is:

SEG$(string,start-position,end-position)

where:

string	is the string from which the segment is returned.

start-position — is a numeric expression specifying the starting character position of the segment.

end-position — is a numeric expression specifying the last character position of the segment.

There are several boundary conditions based on the values of the positions and the string. For details on these boundary conditions, see Book 3.

By using the SEG$ function and the string concatenation operator (& or +), you can replace a segment of a string. For example, line 20 of the following program replaces the characters CDE in the string A$ with XYZ.

```
10 A$ = 'ABCDEFG'
20 C$ = SEG$(A$,1,2)&'XYZ'&SEG$(A$,6,7)
30 PRINT C$
RUNNH

ABXYZFG

READY
```

You can use similar string expressions to replace any given characters in a string. A general formula to replace the characters in positions n through m of string A$ with B$ is:

```
C$ = SEG$(A$,1,n-1)&B$&SEG$(A$,m + 1,LEN(A$))
```

Conversion Functions

MINC provides several string functions that convert characters to numbers and numbers to characters. You can use MINC functions to make the following conversions:

1. Character to its corresponding ASCII code (ASC).
2. ASCII code to its corresponding character (CHR$).
3. Number to its character representation (STR$).
4. Character representation of a number to its numeric value (VAL).
5. String representing a binary number to a decimal number (BIN). (See Book 3 for the description of this function.)
6. String representing an octal number to a decimal number (OCT). (See Book 3 for the description of this function.)

These functions provide you with flexibility in manipulating both strings and numbers.

Character and ASCII Code Conversions

MINC uses the ASCII code to represent characters internally. The ASC function returns the decimal ASCII code of a one-character string that you specify.

See Appendix A for a table of the ASCII codes.

The CHR$ function returns the one-character string that has the ASCII value you specify.

For example:

```
LISTNH
10 PRINT 'The ASCII code for 'A' is ';ASC('A')
20 PRINT 'The ASCII code for 'B' is ';ASC('B')
30 PRINT 'The ASCII code for 'Z' is ';ASC('Z')
40 PRINT 'The ASCII code for 'a' is ';ASC('a')
45 PRINT 'The ASCII code for 'z' is ';ASC('z')
50 PRINT 'The ASCII code';ASC('1');'represents the character ';CHR$(49)
60 PRINT 'The ASCII code';ASC('9');'represents the character ';CHR$(57)

READY
RUNNH

The ASCII code for 'A' is 65
The ASCII code for 'B' is 66
The ASCII code for 'Z' is 90
The ASCII code for 'a' is 97
The ASCII code for 'z' is 122
The ASCII code 49 represents the character 1
The ASCII code 57 represents the character 9

READY
```

The ASC and CHR$ functions can be used with the SEG$ function to analyze the characters in a string.

The form of the ASC function is:

ASC(string)

The string must be a one-character string. If string is a null string or contains more than one character, MINC prints the following error message.

```
?MINC-F-Arguments in definition do not match function called
```

The ASC function produces a whole number value.

The form of the CHR$ function is:

CHR$(numeric expression)

The CHR$ function generates only one character at a time. The numeric expression must be greater than or equal to 0. MINC treats arguments greater than 255 as being modulo 256 (that is, MINC treats 256 as 0, 257 as 1, and so forth).

The numeric expression represents the decimal ASCII value of some characters. Its value is treated modulo 256 because the ASCII codes are within the range 0 to 255.

Numbers and Their String Representation Conversions

Two functions, VAL and STR$, convert numbers to their ASCII character representation and vice versa. You can use these functions to input a numeric value in a string variable or to print a number without the spaces around it.

The VAL function returns the number represented by the specified string. The form of the VAL function is:

VAL(string)

The string may contain the digits 0 through 9, the letter E (for E notation), and the symbols + (plus), - (minus), and . (decimal point), and must be a string representation of a number.

For example:

```
READY
PRINT VAL('23')
  23

READY
PRINT VAL('2.345E6')
  2.34500E+06

READY
A$='12345'

READY
PRINT VAL(A$)
  12345

READY
PRINT VAL('A')

?MINC-F-Arguments in definition do not match function called

READY
```

The STR$ function converts a number to its string representation. The form of the STR$ function is:

STR$(numeric-expression)

The STR$ function returns the value of numeric-expression as it would be printed by a PRINT statement but without a leading or trailing space. Use the STR$ function when you want to print a number without spaces before and after it or when you want to perform string operations or functions on a number.

Example — Converting Lower Case to Upper Case

This example converts lower case input to upper case and leaves upper case input alone. The algorithm for this program can be described as follows:

1. Input a string.
2. Look at the first character in the string.
3. If the character is lower case, then convert it to upper case.
4. Append the converted character on to the new string of upper case.
5. Look at the next character.
6. Go to step 3.

Obviously lines 2 through 6 can be replaced with a FOR loop similar to the unfinished loop shown below.

```
7 FOR I=1 TO length of string
8 T$=next character
9 IF T$ is lower case THEN convert it
10 R$=R$ & T$
11 NEXT I
```

Remember that & and + are the string concatenation operators (see Chapter 3, page 39).

In line 7, *length of string* is simply LEN(S$), where S$ is the string to be converted from lower to upper case.

In line 8, to get the *next character*, use the SEG$ function to extract the *ith* character from the input string as follows.

```
8 T$=SEG$(S$,I,I)
```

So far, the program is this:

```
1 INPUT S$
7 FOR I=1 TO LEN(S$)
```

```
8 T$ = SEG$(S$,I,I)
9 IF T$ is lower case THEN convert it
10 R$ = R$ & T$
11 NEXT I
```

Now all that remains to be done is to determine if T$ is a lower case character and convert it.

To convert a lower case character to an upper case character, subtract the constant 32 from the ASCII code of the lower case character. You do not need to know the ASCII code of any character.

Remember that the function ASC returns the value of the ASCII code of the argument. Thus ASC('a') represents the ASCII code for 'a'. ASC('a')-32 represents the ASCII code for 'A'. Therefore, if T$ is a lower case character, ASC(T$)-32 represents the upper case equivalent of T$. (See Appendix A for the table of ASCII codes.)

In fact, you do not have to remember the constant 32. It turns out that the the following mathematical relationship is true:

```
32 = ASC('a')-ASC('A')
```

To convert T$ from lower case to upper case, you can use the following BASIC statement.

```
T$ = CHR$(ASC(T$)-32)
```

The argument ASC(T$)-32 represents the ASCII code for an upper case character. The function CHR$ converts this ASCII code back to a character.

Now the IF statement looks like this:

IF T$ *is lower case* THEN T$ = CHR$(ASC(T$)-32)

The ASCII code for each of the lower case letters is greater than the ASCII code for the equivalent upper case letters. Therefore, to determine if T$ is lower case, you can use the following relations:

```
ASC(T$)>=ASC('a')

ASC(T$)<=ASC('z')
```

That is, if ASC(T$)>=ASC('a')and if T$<=ASC('z'), then T$ is

a lower case character.

Then, the IF statement becomes:

```
IF ASC(T$)>=ASC('a') THEN IF ASC(T$)<=ASC('z') THEN T$=CHR$(ASC(T$)-32)
```

Finally, the program to convert lower case to upper case input is:

```
10 REM - This program converts lower case string input in S$
20 REM - to upper case, in R$. T$ is a temporary variable.
30 REM - If the string is lower case, it is converted to upper
40 REM - case. If the string is upper case, it is left 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$)>=ASC('a') THEN IF ASC(T$)<=ASC('z')THEN T$=CHR$(ASC(T$)-U)
100 R$=R$ & T$
110 NEXT I
120 PRINT R$
RUNNH

Input a string? yes
YES

READY
```

USER-DEFINED FUNCTIONS

You can define your own functions and then use them as you would the SQR, SIN, or SEG$ functions. For example, the following program defines a function to cube a variable. The function is named FNC, and it is defined in line 10.

```
10 DEF FNC(X)=X^3\ REM — define the cube function
20 PRINT 'I','I^3'
30 FOR I=1 TO 5
40 PRINT I,FNC(I)\ REM — print the number and its cube
50 NEXT I
60 END
RUNNH

I                    I^3
1                    1
2                    8
3                    27
4                    64
5                    125

READY
```

The name of a user-defined function consists of the letters FN followed by a third letter, and optionally followed by a percent

sign (%) or a dollar sign ($). If you end the function name with a percent sign, the function returns an integer; if you end the function name with a dollar sign, it produces a string; otherwise, the function produces a real number. Table 4 shows some valid and invalid user-defined function names.

Table 4. User-Defined Function Names

Valid User-Defined Function Names	*Invalid User-Defined Function Names*
FNA	FN1
FNC%	FNC1
FNR$	FNC%$

You must define each user-defined function once in a program with a DEF statement. You can define it anywhere in the program. The general form of the DEF statement for user-defined functions is:

DEF name (dummy-argument-list) = expression

where:

name	is the function name as described previously.
dummy-argument-list	contains between one and five dummy argument names. A dummy argument in a function definition is a place holder for the actual argument. The arguments may be integer, real, or string variables.
expression	is the string or arithmetic expression that defines what the function is supposed to do. The expression contains any of the dummy arguments or any other variables in the program as well as any MINC-defined function or user-defined function.

You must ensure that the expression is the same type, string or numeric, as the function name. The rules of mixed mode arithmetic apply to user-defined functions.

You can define a function anywhere in the program. You can use any defined function in a program. The format for using the function is:

function-name (actual-argument-list)

For example,

X=FNC(3)

For example, in the following DEF statement, which computes the value of X modulo M, X and M are the dummy arguments.

```
10 DEF FNM(X,M)=X-M*INT(X/M)
```

The variables X and M are place holders for the actual arguments of the function. In the following assignment statement, 53 and 10 are the actual arguments of the function FNM.

```
I=FNM(53,10)
```

You can even use one of the dummy argument names as an actual variable name within the program without affecting the function definition. That is, in this example, you can use variables X and M within the program without affecting the definition of FNM.

For example, in the following program, the value of FNM(53,10) remains unchanged, even though X and M are assigned values in line 30. Note too that the execution of the function does not alter the values of the actual variables X and M.

```
10 DEF FNM(X,M)=X-M*INT(X/M)
20 PRINT FNM(53,10)
30 X=5 \ M=23
40 PRINT FNM(53,10),X,M
RUNNH

 3
 3             5             23
```

The defining expression in a DEF statement can contain any constants, variables, MINC-supplied functions or user-defined functions. For example:

```
10 DEF FNQ(A,B,C)=SQR(B^2-4*A*C)
20 DEF FNR(A,B,C)=(-B-FNQ(A,B,C))/2*A
30 DEF FNS(A,B,C)=(-B+FNQ(A,B,C))/2*A
```

You can include any variables in the defining expression. However, if the expression contains variables that are not in the

dummy argument list, they are not dummy variables. That is, when MINC evaluates the user-defined function, the variables have the value currently assigned to them in the main program.

The definition must have at least one dummy argument. However, the expression does not have to contain any variables. For example:

```
10 DEF FNA$(X) = 'This is a string constant despite the value of X'
20 R$ = FNA$(10)
30 PRINT R$
RUNNH

This is a string constant despite the value of X

READY
```

If you enter a DEF statement in immediate mode, MINC ignores it. MINC does not define a function until a program is run.

For example, if you type a function definition and then try to use the function in the immediate mode, MINC produces an error message.

```
10 DEF FNA(X) = X^2
PRINT FNA(3)

?MINC-F-No DEF statement for the function named

READY
```

However, once you run the program, the function is defined.

```
RUNNH

READY
PRINT FNA(3)
  9

READY
```

Now the function works.

If you try to define the same function more than once in a program, MINC prints the following message.

```
?MINC-F-An earlier statement already defined the function at line 20
```

Line 20 is the line of the second definition, that is, the line with the error.

CHAPTER 9
SUBROUTINES

Some of the previous examples used many BASIC statements to validate input. For example, just validating a yes or no response takes three statements as follows:

```
200 INPUT R$
210 IF R$ = 'YES' GO TO 100
220 IF R$ = 'NO' GO TO 300
230 GO TO 200
```

These three statements do not completely test the input. For example, these three lines do not test for a lower case response or a combination of upper and lower case.

In a long program, you may prompt the user to answer yes or no to more than one question. In this case, you can put the three (or more) lines after every input of yes or no, or you can figure out a way to reuse the same three (or more) validation lines at every yes or no input.

BASIC simplifies reusing the same validation statements with a mechanism called a *subroutine*. A subroutine is a group of statements in a program with distinct entry and exit points that performs a task, such as input validation.

Before the details of using subroutines are discussed, an example is given below to show you how subroutines are used.

This example shows a subroutine that validates a yes or no input. The problem is the questionnaire program from page 101.

The subroutine prints a prompt that the program put in variable P$, checks for a yes or no response, and sets variable Z to 1 for yes and Z to 2 for no.

The essential portion of the previous method for processing questionnaires is:

```
10 DIM R(20,3)
45 PRINT 'Input another questionnaire'; \ INPUT A$
50 IF A$='YES' GO TO 70 \ IF A$='NO' GO TO 310 \ GO TO 45
60 REM - Input and process one questionnaire
70 FOR I=1 TO 20
.
.
.
350 END
```

Below is an example of the essential portion of this program now changed to use a subroutine to validate the yes or no response. The person using the program sees exactly the same thing as in the previous version.

```
10 DIM R(20,3)
45 P$='Input another questionnaire'
50 GOSUB 800
55 ON Z GO TO 70, 310
60 REM - Input and process one questionnaire
70 FOR I=1 TO 20
.
.
.
800 REM - Subroutine to validate yes or no
805 REM - P$ is the prompt string
806 REM - R$ is the response (yes or no)
810 PRINT P$ \ INPUT R$
820 IF R$<>'YES' THEN IF R$<>'NO' GO TO 810
830 IF R$='YES' THEN Z=1
840 IF R$='NO' THEN Z=2
850 RETURN
900 END
```

The GOSUB statement in line 50 transfers control to line 800. The RETURN statement in line 850 returns control to line 55, the line after the GOSUB statement.

This particular example using the subroutine is not shorter than the program it replaced. However, the subroutines handles any case that requires a yes or no response and is therefore more general. Thus, if the program must be expanded to handle many cases with yes or no responses, each new response can be handled by the subroutine.

THE GOSUB AND RETURN STATEMENTS

The GOSUB statement is used to transfer control to a subroutine. The form of the GOSUB statement is:

GOSUB statement-number

The statement-number is the statement number of the first statement in the subroutine to be executed.

For example, the following GOSUB statement transfers control to statement 800.

```
GOSUB 800
```

A GOSUB is not like a GO TO, however, even though it unconditionally transfers control to another statement number. The difference is that when you use a GO TO, MINC unconditionally transfers control to the new statement and forgets the statement number of the GO TO; but when you use a GOSUB, MINC remembers the statement number of the GOSUB statement. Thus, when MINC executes the statement 50 GOSUB 800, it goes to statement 800, but it remembers that it came from statement 50.

The RETURN statement is used to transfer control from the subroutine just executed back to the statement after the GOSUB that called the subroutine.

The form of the RETURN statement is:

```
RETURN
```

The following example expands the use of the validating subroutine. When the subroutine is called from line 110, it returns to line 120. When the subroutine is called from line 210, it returns to line 240.

```
10 REM - This program processes questionnaires
20 REM - the questionnaires are in two groups:
30 REM - the special group, and the rest of the questionnaires
.
.
.
100 P$ = 'Another questionnaire'
110 GOSUB 800
120 ON Z GO TO 200, 500
200 P$ = 'Is questionnaire in special group'
210 GOSUB 800
240 ON Z GO TO 300, 400
300 REM - Process special questionnaires
```

```
.
.
.
400 REM - Process regular questionnaires
.
.
.
500 REM - Print results
.
.
.
799 GO TO 1000
800 REM - Subroutine to validate yes or no response
810 PRINT P$ \ INPUT R$
820 IF R$<>'YES' THEN IF R$<>'NO' GO TO 810
830 IF R$='YES' THEN Z=1
840 IF R$='NO' THEN Z=2
850 RETURN
.
.
.
1000 END
```

Notice that a new statement (number 799) has been put in to go to statement 1000 after printing the results. You must be very careful to make sure that a program does not "fall through" to a subroutine. If statement 799 were not there, control would pass to statement 800 after the results were printed. MINC would finally terminate the program with an error message at statement 850 because MINC would not know where to return.

Because it is difficult to look at a section of a BASIC program and determine whether this section is a subroutine, it is a good idea to begin each subroutine with a REMARK statement that identifies the subroutine.

EXAMPLE OF SUBROUTINES

This section gives more advanced examples of subroutines as well as some techniques and rules.

In this example, the program to convert lower case input to upper case is changed to a subroutine.

Again, the complete program to convert lower to upper case is as follows. (See also Chapter 8, page 130).

```
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, it is left 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 IF ASC(T$)<=ASC('z') THEN T$=CHR$(ASC(T$)-U)
100 R$=R$ & T$
110 NEXT I
120 PRINT R$
```

To make this program a general subroutine, alter the program so that the lower-to-upper-case subroutine no longer prints a message or accepts input. To be most useful and general, the lower-to-upper-case subroutine should assume that the calling program "knows" what it wants converted to upper case. All that a lower-to-upper-case subroutine should do is convert a string from lower to upper case. The program (or other subroutine) that uses the lower-to-upper-case subroutine should worry about printing messages and inputting strings. This subroutine assumes the calling program puts the string to be converted in S$.

Notice in the lower-to-upper-case subroutine shown below that now a new statement (statement 1030) is added to initialize R$ to the null string. Every time you use the RUN command to run the program to convert lower case to upper case, MINC reinitializes all numeric variables to 0 and all string variables to the null string (''). However, when writing a subroutine, you do not know how many times the subroutine will be used within one run of the program. Thus, during one run, R$ contains the old string from the previous time it was used, unless you remove the old string.

```
1000 REM - Subroutine to convert lower to upper case
1010 REM - Subroutine converts only lower case
1020 U=ASC('a')-ASC('A')
1030 R$=''
1040 FOR I=1 TO LEN(S$)
1050 T$=SEG$(S$,I,I)
1060 IF ASC(T$)>=ASC('a') THEN IF ASC(T$)<=ASC('z') THEN T$=CHR$(ASC(T$)-U)
1070 R$=R$ & T$
1090 NEXT I
1100 RETURN
```

The new version of the yes-no validation subroutine that uses the lower-to-upper-case subroutine is as follows:

```
800 REM - Subroutine to validate yes or no response
810 PRINT P$ \ INPUT S$
820 GOSUB 1000 \ REM - Convert response to upper case
830 IF R$<>'YES' THEN IF R$<>'NO' GO TO 810
840 IF R$='YES' THEN Z=1
850 IF R$='NO' THEN Z=2
860 RETURN
```

Notice that the yes-no validation subroutine now uses another subroutine. This process is called *nesting subroutines.*

You must remember, that any program that uses the yes-no validation subroutine given here must place a prompt in P$ and must not use variables S$, R$, T$, or U. That is, unlike user-defined functions that have dummy arguments, variables used in subroutines are actual variables within the program.

THE ON/GOSUB STATEMENT

The ON/GOSUB statement is used to conditionally transfer control to one of several subroutines. The ON/GOSUB statement has the following form:

ON numeric-expression GOSUB list-of-statement-numbers

The numeric-expression is any valid numeric expression and the statement numbers must be separated by commas.

The ON/GOSUB statement works like the ON/GO TO statement (see page 82). When MINC executes the ON/GOSUB statement, it first evaluates the numeric expression. If the value of the expression is 1, control passes to the first line number specified; if it is 2, control passes to the second line number specified; and so forth. If the expression is less than 1 or greater than the number of line numbers in the list, MINC prints the following error message.

```
?MINC-F-Value of control expression is out of range at line XX
```

where XX represents the statement number where the error occurred.

The following statement is an example of an ON/GOSUB statement.

```
20 ON A+B GOSUB 200,300,120
```

If A+B is equal to 1, then MINC passes control to line 200. If A+B is equal to 2, then MINC passes control to line 300. If A+B is equal to 3, then MINC passes control to line 120. If A+B is greater than 3 or less than 1, MINC prints out the error message and stops executing the program.

In this example, no matter which subroutine is executed from the ON/GOSUB statement, as soon as MINC reaches a RETURN statement, it passes control back to the statement

physically after the ON/GOSUB statement (the statement after line 20 in this example).

RESEQUENCING PROGRAMS WITH GOSUBS

When you use the RESEQ command to resequence all or part of a program, MINC correctly resequences all references to subroutines. (See pages 82 and 83 for other references to the RESEQ command.) For example, suppose the program below is in need of resequencing.

```
3 DIM R(20,3)
6 P$ = 'input another questionnaire'
9 GOSUB 45
12 ON Z GO TO 15,30
15 FOR I = 1 TO 20
18 PRINT I; \ INPUT Q
21 R(I,Q) = R(I,Q) + 1
24 NEXT I
27 GO TO 6
30 PRINT ,,'AGREE','DISAGREE','DON'T CARE'
33 FOR I = 1 TO 20
36 PRINT 'question';I,,R(I,1),R(I,2),R(I,3)
39 NEXT I
42 GO TO 63
45 REM - subroutine to validate yes or no
48 PRINT P$ \ INPUT R$
51 IF R$<>'yes' THEN IF R$<>'no' GO TO 48
54 IF R$ = 'yes' THEN Z = 1
57 IF R$ = 'no' THEN Z = 2
60 RETURN
63 END
```

Then, the RESEQ command changes the statement numbers, and the program is numbered as shown below. Notice that the GO TOs, the ON/GO TOs, and the GOSUB statements are renumbered correctly.

```
10 DIM R(20,3)
20 P$ = 'Input another questionnaire'
30 GOSUB 150
40 ON Z GO TO 50,100
50 FOR I = 1 TO 20
60 PRINT I; \ INPUT Q
70 R(I,Q) = R(I,Q) + 1
80 NEXT I
90 GO TO 20
100 PRINT ,,'AGREE','DISAGREE','DON'T CARE'
110 FOR I = 1 TO 20
120 PRINT 'question';I,,R(I,1),R(I,2),R(I,3)
130 NEXT I
140 GO TO 210
150 REM - subroutine to validate yes or no
```

```
160 PRINT P$ \ INPUT R$
170 IF R$<>'yes' THEN IF R$<>'no' GO TO 160
180 IF R$='yes' THEN Z=1
190 IF R$='no' THEN Z=2
200 RETURN
210 END
```

MINC can also resequence programs with ON/GOSUB statements. Remember, however, that the RESEQ command does not update any statement numbers that are referenced within REMARK statements.

CHAPTER 10
MINC ROUTINES

You can use the graphic, instrument bus, and lab module features of MINC in the immediate mode or in the program mode. The MINC system provides these capabilities in *routines*, a means by which you request MINC to perform a complex task.

There is a set of graphic routines, a set of instrument bus routines, and a set of lab module routines. Book 4 describes how to use the graphic routines, Book 5 describes how to use the instrument bus routines, and Book 6 describes how to use the lab module routines.

This chapter describes the general format for using routines. The examples in this chapter use graphic routines because all MINC systems include the graphic capability, but the concepts described here apply to the lab module and instrument bus routines as well.

For example, the following BASIC program uses the graphic routine HTEXT to display two lines on the screen, a flashing boldface line and a reverse video line. Type this program in and run it to see the graphics that this program generates.

```
10 HTEXT('flash,bold',22,1,'Flashing boldface')
20 HTEXT('reverse',23,1,'Reverse video')
```

HTEXT is the *routine name*, 'flash,bold', 22, 1, and 'Flashing boldface' are the routine arguments.

The task that routine HTEXT performs is printing a line on the terminal screen. The way the printed line looks depends on the arguments that you specify.

As you can see by this example, you use a routine as you would any other BASIC statement. You can use routine statements where you need them in a program along with any other BASIC statements. For example:

```
10 PRINT 'Another line'; \ INPUT R$
20 IF R$<>'yes' THEN STOP
30 PRINT 'Input line to print in flashing boldface'; \ INPUT L$
40 IF L$<>'' THEN htext('flash,bold',23,1,L$) \ GO TO 10
50 PRINT 'Line entered is null —— try again' \ GO TO 30
```

Notice that MINC does not convert routine names to upper case as it does for any other BASIC statement. Thus, if you type in a routine name in lower case, it remains in lower case.

For more information on the routines, see Books 4, 5, and 6.

CHAPTER 11
FILE CONTROL

When you type the SAVE command, MINC saves the program in the workspace in a program file on the volume that you specify. In Chapter 4 the definition of program file is simply that place on the volume in which the program is stored.

A *file* in general is a named portion of a volume that holds information. It is conceptually similar to a file kept in a drawer in a filing cabinet. You can think of the cabinet as a volume, and the drawers as files. Each folder within the drawer holds information (probably on paper). The drawer (file) holds a set of similar data. Each folder holds one record or entry. However, the kind of information in a file may vary from file to file. For example, one file may hold correspondence and another file may hold sales brochures.

A file on a volume can hold a program. Other files on the same volume can hold, for example, the ASCII characters of a letter you typed or data collected by an instrument connected to MINC. This chapter discusses the two types of MINC files, sequential files and virtual array files, and suggests possible uses for these files.

PROGRAM FILES AND OTHER FILES

When you type a SAVE command, MINC creates a file on the volume you specify with the name that you specify. For example:

```
SAVE QUEST
```

This command creates a program file on the default volume,

names the program file QUEST.BAS, and stores the ASCII characters of the program in the workspace into the program file. The program file on the volume looks like this:

10 ⓢⓟ DIM ⓢⓟ R(20,3) ⓡⓔⓣ 20 ⓢⓟ P$ = 'Input ⓢⓟ another ⓢⓟ quest...

That is, the program file holds every ASCII character in the program, including the spaces (ⓢⓟ) and RETURNs (ⓡⓔⓣ). When you use the OLD command, MINC automatically transfers the ASCII characters from the file into the workspace. Then the program is there, ready for you to run.

There are other kinds of files besides program files. For example, you may want to collect more data from an instrument than MINC can store in the workspace. To do this, you must create a file on a volume in which MINC can store the data. As another example, you may need to use an array in a program that does not fit in the workspace. You can create a file on a volume that MINC can use as extra array space.

When you are using program files, MINC manages them for you when you type the SAVE, REPLACE, or OLD commands. When you use nonprogram files, such as a data file, you must handle creating the file, putting information into the file, and storing the file yourself with the appropriate BASIC statements in a BASIC program.

To create a file or use an existing file, you must *open* the file with an OPEN statement. To make sure the file is stored properly on the volume, you must *close* the file with a CLOSE statement after you have used the file. How you use the file depends on the type of file; that is, whether it is a sequential file or a virtual array file.

A *sequential file* is a file that must be accessed serially. That is, to get to the fifth piece of information in the file, MINC must first look at pieces one, two, three, and four. The information in a sequential file is always ASCII characters, and MINC can either output information from the workspace to an open sequential file or input information from a sequential file to the workspace, but MINC cannot do both input and output with the same sequential file. Program files are sequential files.

A *virtual array file* is a file that can be accessed directly. That is, MINC can access the fifth piece of information without having to access the first four. This type of file is called a virtual *array* file, because you use it in your program as you would an array.

The term *virtual* is used because this file is not an array stored in the workspace even though it looks like an array to the program; it is a file stored on a volume. Because you can treat a virtual array file like an array, MINC can both input from and output to the same virtual array file.

The information in a virtual array file can be ASCII characters or it can be numeric information where 2 is stored as the number 2 and not as the ASCII character "2".

The following sections describe using sequential and virtual files.

SEQUENTIAL FILES

You use sequential files in the same way that you use terminal input and output. However, when you input from a sequential file (using an INPUT or LINPUT statement), MINC inputs the information from the sequential file instead of from the user sitting at the terminal. When you output to a sequential file (using a PRINT statement), MINC writes the information in the file instead of on the terminal screen. Before you can use a sequential file, however, you must open the file.

Opening a Sequential File

You open a sequential file with an OPEN statement. The OPEN statement opens a *channel* over which MINC transfers the information from the file to the workspace. A channel in MINC is similar to a television channel. Currently, television channels are open for input only. You tune your television to a channel and receive a television program.

The OPEN statement links the file with the channel until the channel is closed by a CLOSE statement. Then the channel becomes available again. There are 12 available channels numbered 1 through 12. Channel 0 is always open to the terminal, and you cannot open it.

The three most common forms of the OPEN statement are:

```
OPEN filespec-string FOR INPUT AS FILE # numeric-expression
```

```
OPEN filespec-string FOR OUTPUT AS FILE # numeric-expression
```

and

```
OPEN filespec-string AS FILE # numeric-expression
```

where:

filespec-string	is a string representing the file specification. The file name is any name (up to 6 characters) that you choose to call the file. The file type is any 3-character file type you want. If you do not specify the file type, MINC defaults to .DAT for the file type. The .DAT file type stands for data. Thus the file type is by default not a program file.
FOR INPUT	is optional and specifies using an existing file.
FOR OUTPUT	specifies creating a new file.
#	is optional.
numeric-expression	is the channel number of the file. Later in the program when you want to refer to the file, you use the channel number instead of the file name. The channel number can have any whole number value between 1 and 12.

If you specify FOR INPUT, MINC opens an existing file, and you can only input information from it to the workspace.

If you specify FOR OUTPUT, MINC creates a new file. Any existing file with the same file specification is superseded when the new file is closed (see next section). If you specify FOR OUTPUT for a sequential file, you can only write to the file.

Specifying neither FOR INPUT nor FOR OUTPUT for an existing sequential file is equivalent to specifying FOR INPUT. If the sequential file does not exist, specifying neither is equivalent to specifying FOR OUTPUT.

If you use LP: as the filespec with the FOR OUTPUT option, MINC can output to the line printer with the PRINT# statement described later in the chapter. For more details see the section entitled "Line Printer" in Book 3.

Following is an example of creating a new sequential file called QIN.DAT on the system volume.

```
10 OPEN 'QIN' FOR OUTPUT AS FILE #1
```

Remember that the .DAT file type is the default file type. If you want to create a file with another file type, you must specify the file type in the file specification. For example, the following OPEN statement creates a new sequential file with the name QIN.TXT on the SY1: volume.

```
10 OPEN 'SY1:QIN.TXT' FOR OUTPUT AS FILE #2
```

For more advanced features of the OPEN statement, see Book 3.

Closing a Sequential File

When you create a sequential file using the OPEN statement with FOR OUTPUT, you must close the file in the program with a CLOSE statement. If you do not close the file and the program terminates with an error, MINC does not create the file.

The two forms of the CLOSE statement are:

CLOSE # expression, # expression, ...

CLOSE

The expression is the channel number of the file to close. You can close more than one file with the CLOSE statement.

If you use the CLOSE statement with no arguments, MINC closes all open files.

If the program terminates normally (that is, by executing the END statement or the statement with the highest line number), MINC closes all files for you automatically. However, you should put in a CLOSE statement.

Using a Sequential File

You use sequential files in the same way that you use terminal input and output. You program MINC to input from a sequential file with an INPUT or LINPUT statement, and MINC inputs the information from the sequential file instead of from the terminal. You put information into the sequential file with a PRINT # statement.

Storing Data in a Sequential File

You create a sequential file by opening with the FOR OUTPUT specification. Once the file is open, you put information into it with the PRINT # statement.

The form of the PRINT # statement is:

PRINT # expression,argument-list

where:

expression	is the channel number. Note that this channel number must match a channel number from an OPEN statement. If the expression is 0, MINC prints the output on the terminal.
argument-list	contains the items to be printed just as you would print information on the terminal screen. The argument list can contain any numeric and string expressions and the TAB function. You can separate the items to be printed with commas or semicolons, directing MINC to output the information to the file with or without print zones.

You can use a colon (:) instead of a comma (,) after the expression.

If there are no items in the argument list, MINC prints a blank line in the file. When there are no items in the list, you need not specify the comma or colon after the expression.

The following example creates a sequential file and puts 5 names in it.

```
10 OPEN 'NAMES' FOR OUTPUT AS FILE #1
20 PRINT #1, 'JOHN S. DOE';','; 'THOMAS R. SMITH'
30 PRINT #1,'JANET Q. BROWN';',';' CHERYL F. JONES'
40 PRINT #1, 'ANDREW G. SCOTT'
50 CLOSE #1
60 END
RUNNH

READY
```

Line 10 creates the file. Lines 20, 30, and 40 put the names in the file. Notice that the #1 in the PRINT statements matches the #1 in the OPEN statement.

Notice also that between each name is a comma in a string literal (","). You should print a comma in a string literal between each data item on a line in the file. If you do not, later you will not be able to input the data items from the file using an INPUT statement.

The CLOSE statement in line 50 closes the file.

After you run this program, if you look at the directory (DIR), you will see a new file on the volume called NAMES.DAT. If you look at NAMES.DAT with the TYPE command, the file looks like this:

```
JOHN S.DOE,THOMAS R. SMITH
JANET Q. BROWN,CHERYL F. JONES
ANDREW G. SCOTT
```

Accessing Data in a Sequential File

The following example reads the file NAMES.DAT created in the previous program, and prints the results on the terminal.

```
10 OPEN 'NAMES' FOR INPUT AS FILE #1
20 INPUT #1,A$,B$
30 LINPUT #1,C$
40 INPUT #1,D$
50 PRINT A$,B$
60 PRINT C$
70 PRINT D$
80 END
RUNNH

JOHN S. DOE              THOMAS R. SMITH
JANET Q. BROWN,CHERYL F. JONES
ANDREW G. SCOTT
```

Notice that the first INPUT # statement at line 20 has the same number of variables (A$,B$) as the first line in the NAMES file has values (JOHN S. DOE, THOMAS R. SMITH). An INPUT # statement uses one whole line of input, even if the number of variables in the statement is less than the number of values in the corresponding line of the input file. For example, suppose the input line was A,B,C and the input statements were:

```
10 INPUT #1, A$,B$
20 INPUT #1, C$
```

The value of A$ then becomes "A" and the value of B$ becomes "B". However, the value of C$ becomes '' (the null string). The first INPUT # statement used the whole line of input, despite the fact that there were fewer variables than values.

The LINPUT # statement inputs a string from a file. MINC treats LINPUT # just as it treats LINPUT; all characters on the input line, including commas and quotation marks, are assigned to the string. Like the INPUT # statement, the LINPUT # statement expects input from the terminal if the file number is 0. The LINPUT # statement in line 30 reads the entire line of input from the NAMES file, including the comma.

Checking for the End of the Input File

Suppose that you have the following input file.

JOHN S. DOE
JANET Q. BROWN
ANDREW G. SCOTT
THOMAS R. SMITH
CHERYL F. JONES

You could then write the following program to read the file and write out its contents.

```
10 OPEN 'NAMES' FOR INPUT AS FILE #2
20 LINPUT #2,N$
30 PRINT N$
40 GO TO 20
50 CLOSE #2
60 END
RUNNH
JOHN S. DOE
JANET Q. BROWN
ANDREW G. SCOTT
THOMAS R. SMITH
CHERYL F. JONES

?MINC-F-Too few values for INPUT or READ variables at line 20
```

Notice that this program terminates with an error message. MINC continued to read the file until it came to the end. When MINC reached the end of the file, there were no more values to read with the LINPUT # statement, so MINC terminated the program and printed the message.

In this case, the program is terminated before the file is closed in line 50. There are no problems in this example, because the program does not create any sequential files. However, if the program were creating a sequential file, and did not close the file, the file would be lost. You can use the CLOSE statement in the immediate mode if this happens (see Book 3).

You can check for the end of file and have MINC terminate the program without an error by using the IF END # statement. The form of this statement is:

IF END # expression THEN statement
IF END # expression THEN statement-number
IF END # expression GO TO statement-number

The expression is the file number of the file. The value of the expression can not be 0 and the file associated with the expression cannot be a terminal.

If the next attempt to input a value would produce the ?MINC-F-Too few values for INPUT or READ variables at line XX error message, MINC executes the statement after the THEN or transfers control to the specified line number. Otherwise MINC transfers control to the statement after the IF as it does with any other IF statement.

The following example terminates without an error message.

```
10 OPEN 'NAMES' FOR INPUT AS FILE #2
20 IF END #2 GO TO 40
30 GO TO 60
40 PRINT 'End of file'
50 GO TO 90
60 LINPUT #2, N$
70 PRINT N$
80 GO TO 20
90 CLOSE #2
100 END
RUNNH

JOHN S. DOE
JANET Q. BROWN
ANDREW G. SCOTT
THOMAS R. SMITH
CHERYL F. JONES
End of file

READY
```

Line 20 checks for the end of the file.

Note that the IF END # statement tests if there is one more item in the file. If there is one item left when the IF END checks the file and your INPUT # statement requests two items, MINC prints the ?MINC-F-Too few values for INPUT or READ variables at line XX error message and terminates the program without closing the file.

Restoring a File to the Beginning

The RESTORE # statement resets the specified sequential input file from its current position to its beginning. The format of the RESTORE # statement is:

RESTORE # expression

The expression is the channel number of the file to be restored.

Example of Using Sequential Files

This example merges two files of names and addresses in alphabetical order into one file in alphabetical order and removes duplicate names and addresses.

The algorithm description of the basic loop is:

1. Input a name and address (name1) from file 1 and a name and address (name2) from file 2.

2. If name1 < name2 then put name1 in the output file, get next name1 from file 1, and repeat step #2.

3. If name2 < name1 then put name2 in the output file, get next name2 from file 2, and go to step #2.

4. At this point name1 = name2. If addresses match then they are duplicates, put name1 in file, and go back to step #1.

5. If addresses do not match, put both names and addresses in file and go back to step #1.

The following section of a BASIC program implements this algorithm. Compare this program section to the algorithm. N1$ is the name from file number 1, A1$ is the street or P.O. Box number from number 1, and S1$ is the city and state from file number 1. N2$,A2$, and S2$ correspond to the names, streets or P.O. Box numbers, and cities from file number 2. File number 3 is the merged output file.

Lines 95 through 320 correspond to step 1 of the algorithm. Lines 330 through 400 correspond to step 2 of the algorithm. Lines 405 through 460 correspond to step 3 of the algorithm. Lines 465 through 510 correspond to step 4 of the algorithm. Finally, lines 515 through 560 correspond to step 5 of the algorithm.

```
95 REM -------------------------
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$

195 REM -------------------------
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350

305 REM -------------------------
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
```

```
330 REM ------------------------
340 REM - Compare the files
345 REM
350 IF SEG$(N1$,1,28)>=SEG$(N2$,1,28) GO TO 410
355 REM ------------------------
360 REM - File 1 alphabetically precedes 2
365 REM
370 PRINT #3,N1$
380 PRINT #3,A1$
390 PRINT #3,S1$
400 GO TO 230
405 REM ------------------------
406 REM - Compare the files
407 REM
410 IF N2$=N1$ GO TO 480
415 REM ------------------------

420 REM - Record from file 2 alphabetically precedes 1
425 REM
430 PRINT #3,N2$
440 PRINT #3,A2$
450 PRINT #3,S2$
460 GO TO 120
465 REM ------------------------
470 REM - The names are equal
475 REM
480 PRINT #3,N1$
490 PRINT #3,A1$
500 PRINT #3,S1$
505 REM ------------------------
506 REM - Compare the addresses
507 REM
510 IF A1$=A2$ THEN IF S1$=S2$ THEN GO TO 60
515 REM ------------------------
520 REM - the addresses are different, so print both
525 REM
530 PRINT #3,N2$
540 PRINT #3,A2$
550 PRINT #3,S2$
560 GO TO 60
```

This program segment is not complete. It does not open the files nor test for the end of either of the files. First, in this example, the program must terminate normally or the new, merged file will not be closed, and consequently will not exist. Second, if one of the files is ended and the other is not, the program should put the rest of the unfinished file on the end of the merged file.

A specific example of the way the program should work is shown below. Notice that file 2 is longer than file 1.

File 1

```
DIGITAL EQUIPMENT CORP. A/S
P.O. Box 3914
7001 Trondheim, Norway
DIGITAL EQUIPMENT CORP. S.A.
Burgunderstrasse 42A
Basle, Switzerland CH-4051
DIGITAL EQUIPMENT INT., LTD.
Ballybrit Industrial Estate
Galway, Ireland
```

File 2

```
DEC DE PUERTO RICO
P.O. Box 106
San German, Puerto Rico 00753
DIGITAL EQUIPMENT CORP.
1400 Terra Bella Avenue
Mountain View, CA 94043
DIGITAL EQUIPMENT CORP.
200 Forest Street
Marlboro, MA 01752
DIGITAL EQUIPMENT CORP.
P.O. Box 80
Albuquerque, NM 87103
DIGITAL EQUIPMENT INT., LTD.
Ballybrit Industrial Estate
Galway, Ireland
```

Merged File

```
DEC DE PUERTO RICO
P.O. Box 106
San German, Puerto Rico 00753
DIGITAL EQUIPMENT CORP.
1400 Terra Bella Avenue
Mountain View, CA 94043
DIGITAL EQUIPMENT CORP.
200 Forest Street
Marlboro, MA 01752
DIGITAL EQUIPMENT CORP.
P.O. Box 80
Albuquerque, NM 87103
DIGITAL EQUIPMENT CORP. A/S
P.O. Box 3914
7001 Trondheim, Norway
DIGITAL EQUIPMENT CORP. S.A.
Burgunderstrasse 42A
Basle, Switzerland CH-4051
DIGITAL EQUIPMENT INT., LTD.
Ballybrit Industrial Estate
Galway, Ireland
```

The end conditions for this problem are rather complex. You do not want one of the files to finish causing the LINPUT statement to run out of data. If this happens, MINC terminates the program abnormally with an error message. Not only is the merged file not closed, it also is not finished. The names and addresses in the longer file do not appear in it because the program terminated before it read them. The LINPUT # statements given previously are:

```
110 LINPUT #1,N1,A1$,S1$
210 LINPUT #2,N2$,A2$,S2$
320 LINPUT #1,N1$,A1$,S1$
```

When the LINPUT # statement at line 110 inputs the last value from file 1, the program must still finish processing file 2.

The BASIC statements to do this are:

```
40 REM ------------------------
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
```

```
70 GO TO 110
80 F=2
90 GO TO 590
95 REM------------------------
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$

195 REM------------------------
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350

305 REM------------------------
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
.
.
.

580 REM - Finish off remaining file
585 REM
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$\PRINT #3,A$\PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END
```

When control passes to line 210, the program has already input a name and address from file 1 (N1$,A1$, and S1$). However, if file 2 has no more values, then the program should print out N1$, A1$, and S1$ before finishing off file 1 at line 590. Otherwise, program control still passes to line 590, but the program ignores the values left in N1$, A1$, and S1$.

The BASIC statements to do this are:

```
40 REM------------------------
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
70 GO TO 110
80 F=2
90 GO TO 590
95 REM------------------------
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$
115 REM------------------------
116 REM - Check for end of file 2
```

```
117 REM
120 IF END #2 GO TO 140
130 GO TO 210
140 F=1
145 REM ------------------------
150 REM - End of file 2, finish file 1
155 REM
160 N$=N1$
170 A$=A1$
180 S$=S1$
190 GO TO 610
195 REM ------------------------
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350

305 REM ------------------------
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
.
.
.
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$\PRINT #3,A$\PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END
```

Finally, when control passes to line 320, the program has already input a name and address from file 2. Therefore, if file 1 has no more values, then the program must print out the current value of N2$,A2$, and S2$ before returning to line 590 and finishing file 2.

The BASIC statements to do this are:

```
40 REM ------------------------
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
70 GO TO 110
80 F=2
90 GO TO 590
95 REM ------------------------
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$
115 REM ------------------------
116 REM - Check for end of file 2
117 REM
```

```
120 IF END #2 GO TO 140
130 GO TO 210
140 F=1
145 REM -------------------------
150 REM - end of file 2, finish file 1
155 REM
160 N$=N1$
170 A$=A1$
180 S$=S1$
190 GO TO 610
195 REM -------------------------
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350
225 REM -------------------------
226 REM - Check for end of file 1
227 REM
230 IF END #1 GO TO 250
240 GO TO 320
245 REM -------------------------
250 REM - End of file 1, finish file 2
255 REM
260 F=2
270 N$=N2$
280 A$=A2$
290 S$=S2$
300 GO TO 610
305 REM -------------------------
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
.
.
.
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$\PRINT #3,A$\PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END
```

The whole BASIC program to merge two files is:

```
10 OPEN 'NAMES1' FOR INPUT AS FILE #1
20 OPEN 'NAMES2' FOR INPUT AS FILE #2
30 OPEN 'MAILST' FOR OUTPUT AS FILE #3
40 REM -------------------------
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
70 GO TO 110
80 F=2
90 GO TO 590
```

```
95 REM - - - - - - - - - - - - - - - - - - - - - - - -
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$
115 REM - - - - - - - - - - - - - - - - - - - - - - - -
116 REM - Check for end of file 2
117 REM
120 IF END #2 GO TO 140
130 GO TO 210
140 F=1
145 REM - - - - - - - - - - - - - - - - - - - - - - - -
150 REM - End of file 2, finish file 1
155 REM
160 N$=N1$
170 A$=A1$
180 S$=S1$
190 GO TO 610
195 REM - - - - - - - - - - - - - - - - - - - - - - - -
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350
225 REM - - - - - - - - - - - - - - - - - - - - - - - -
226 REM - Check for end of file 1
227 REM
230 IF END #1 GO TO 250
240 GO TO 320
245 REM - - - - - - - - - - - - - - - - - - - - - - - -
250 REM - End of file 1, finish file 2
255 REM
260 F=2
270 N$=N2$
280 A$=A2$
290 S$=S2$
300 GO TO 610
305 REM - - - - - - - - - - - - - - - - - - - - - - - -
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
330 REM - - - - - - - - - - - - - - - - - - - - - - - -
340 REM - Compare the files
345 REM

350 IF SEG$(N1$,1,28)>=SEG$(N2$,1,28) GO TO 410
355 REM - - - - - - - - - - - - - - - - - - - - - - - -
360 REM - File 1 alphabetically precedes 2
365 REM
370 PRINT #3,N1$
380 PRINT #3,A1$
390 PRINT #3,S1$
400 GO TO 230
405 REM - - - - - - - - - - - - - - - - - - - - - - - -
406 REM - Compare the files
407 REM
410 IF N2$=N1$ GO TO 480
```

```
415 REM ------------------------
420 REM - Record from file 2 alphabetically precedes 1
425 REM
430 PRINT #3,N2$
440 PRINT #3,A2$
450 PRINT #3,S2$
460 GO TO 120
465 REM ------------------------
470 REM - the names are equal
475 REM
480 PRINT #3,N1$
490 PRINT #3,A1$
500 PRINT #3,S1$
505 REM ------------------------
506 REM - Compare the addresses
507 REM
510 IF A1$=A2$ THEN IF S1$=S2$ THEN GO TO 60
515 REM ------------------------
520 REM - The addresses are different, so print both
525 REM
530 PRINT #3,N2$
540 PRINT #3,A2$
550 PRINT #3,S2$
560 GO TO 60
570 REM ------------------------
580 REM - Finish off remaining file
585 REM
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$ \ PRINT #3,A$ \ PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END
```

This program is still not foolproof. A name typed in upper and lower case is not equal to the same name typed in upper case only. To make sure you are comparing comparable names, you can use the lower-to-upper-case subroutine.

Also, the LINPUT statements input three lines from a name and address file. If the number of lines in each input file is not a multiple of three, then the program will terminate abnormally and the output file will not be created. You can correct this problem by inserting an IF END statement before inputting each variable.

You should now type this program and save it. You cannot test this program, however, until you create two files of names and addresses named NAMES1 and NAMES2.

The following example shows you how to create these name and

address files as well as shows you the general form of the OPEN statement.

Another Example

This example allows you to create a sequential file and put information into the file by running the program. (Note: You can also create files using the editor explained in Chapter 15.) However, this program does not let you correct the file if you make a mistake in typing. That is, if you press RETURN before you notice the mistake, you cannot correct the file with this program. You can, of course, use the DELETE key if you notice the mistake before you press RETURN.

```
10 PRINT "File number(1-12)"; \ INPUT N
20 PRINT "File name(max. of 6 characters)"; \ LINPUT F$
30 PRINT \ PRINT
40 PRINT "At the question mark, type the next line of the file."
50 PRINT "To end, press RETURN at the question mark."
60 REM - - Create arrays to hold records
70 DIM A$(100)
80 DIM B$(100)
90 DIM C$(100)
100 REM - - Accept records from terminal
110 T=-1
120 FOR I=0 TO 100
130 LINPUT L$
140 IF L$="" THEN GO TO 330
150 LINPUT M$
160 LINPUT N$
170 REM - - Change lower case in name to upper case
180 U=ASC("a")-ASC("A")
190 R$=""
200 FOR K=1 TO LEN(L$)
210 T$=SEG$(L$,K,K)
220 IF ASC(T$)>=ASC("a") THEN IF ASC(T$)<=ASC("z") THEN T$=CHR$(ASC(T$)-U)
230 R$=R$&T$
240 NEXT K
250 L$=R$
260 REM - - Insert records into arrays
270 A$(I)=L$
280 B$(I)=M$
290 C$(I)=N$
300 T=T+1
310 NEXT I
320 REM - - Open output file
330 OPEN F$ FOR OUTPUT AS FILE #N
340 REM - - Arrange records in alphabetical order
350 FOR C=0 TO T
360 FOR D=C TO T
370 IF SEG$(A$(C),1,28)>SEG$(A$(D),1,28) THEN GO TO 460
380 NEXT D
390 REM - - Send record to output file
400 PRINT #N,A$(C)
410 PRINT #N,B$(C)
420 PRINT #N,C$(C)
```

```
430 NEXT C
440 CLOSE #N \ GO TO 500
450 REM - - Switch positions of two records
460 S$ = A$(C) \ A$(C) = A$(D) \ A$(D) = S$
470 S$ = B$(C) \ B$(C) = B$(D) \ B$(D) = S$
480 S$ = C$(C) \ C$(C) = C$(D) \ C$(D) = S$
490 GO TO 380
500 PRINT \ PRINT
510 PRINT "The file ";F$;" is created."
520 END

SAVE INFILE

READY

RUNNH

File number (1-12)?1
File name (max. of 6 characters)? NAMES1

At the question mark, type the next line of the file.
To end, press RETURN at the question mark.

?Digital Equipment Int., Ltd.
?Ballybrit Industrial Estate
?Galway, Ireland
?Digital Equipment Corp. S.A.
?Burgunderstrasse 42A
?Basle, Switzerland CH-4051
?Digital Equipment Corp. A/S
?P.O. Box 3914
?7001 Trondheim, Norway
?(RET)
The file NAMES1 is created.

READY
```

To see the file, use the TYPE command. For example,

```
TYPE NAMES1.DAT
```

Remember that you must specify the .DAT file type because the TYPE command defaults to .BAS.

Now you can create the name and address files and run the file merge program.

VIRTUAL ARRAY FILES

You can use virtual array files in the same way that you use a large array. Just as you can access the elements of an an array in the workspace in any order, you can access the elements of a virtual file in any order.

Virtual array files have several advantages over sequential files.

- You can access elements in a direct, nonsequential manner. The last element in a virtual array file can be accessed as quickly as any other element. Remember that when using a sequential file, you must input the entire file before inputting the last element.

- When MINC stores data in virtual array files, it does not convert them to ASCII characters but rather stores them in their numeric representation. Consequently, there is no loss of precision caused by data conversion. There is some loss of precision with sequential files because all data are converted to ASCII. Remember that sequential files are ASCII files only.

- You can update virtual array files without copying the entire file. That is, you can open a virtual array file for both input and output.

Virtual array files also have advantages over arrays stored in the workspace.

- Virtual array files allow you to create much larger arrays than can be stored in the workspace.

- You can permanently store data in virtual array files. That is, when your program ends, the virtual array files are stored on a volume. Remember that when you run a program or use the SCR or CLEAR commands, previous arrays stored in the workspace are cleared, and thus lost.

Virtual array files also have several restrictions that do not apply to arrays stored in the workspace.

- Virtual array files are slower because MINC must read the file on the volume before manipulating data.

- Although strings stored in the workspace can have length (up to 255 characters), you cannot use these variable-length strings in virtual array files. Strings in virtual array files have a fixed maximum length from 1 to 255 that you specify in the DIM # statement (explained in the next section). MINC handles strings shorter than this maximum length similarly to strings

stored in the workspace. MINC truncates strings longer than the maximum length.

- You must dimension only one virtual array file in each DIM # statement.

- Many MINC arrays do not allow virtual array files as arguments although they accept workspace arrays as arguments.

Opening a Virtual Array File

As with sequential files, you open a virtual array file with an OPEN statement. The OPEN statement opens a sequential file unless the file-channel number specified in the OPEN statement is also specified in a DIM # statement (explained later in this section). Again, the three most common forms of the OPEN statement are:

```
OPEN filespec-string FOR INPUT AS FILE # numeric-expression
```

```
OPEN filespec-string FOR OUTPUT AS FILE # numeric-expression
```

and

```
OPEN filespec-string AS FILE # numeric-expression
```

where:

filespec	is a string representing the file specification. If you do not specify the file type, MINC defaults to .DAT for the file type.
FOR INPUT	is optional and specifies opening an existing file.
FOR OUTPUT	specifies creating a new file.
#	is optional.
numeric expression	is the channel number of the file. Later in the program when you want to refer to the file, you use the number instead of the file name. The number can have any whole number value from 0 to 12. The channel number must match the channel number in the corresponding DIM# statement.

variable-name (number-of-elements)

where variable-name can be any string, integer, or real variable name, and number-of-elements represents the dimensions (maximum size) of the subscript or subscripts.

string size — is an optional whole number literal (with or without a percent sign) that specifies the maximum length for elements in a string virtual array file. Its value must be in the range 1 to 255. If it is omitted for a string array, the maximum length is 16.

To access information in an existing virtual array file, be sure that the DIM # statement specifies the same variable type (string, real, or integer) and number of subscripts that are specified in the program that created the file. The variable name associated with the file can be different from the original as long as it is the same variable type.

Below are some examples of opening virtual array files.

The following example creates a 2001-element integer virtual array file. The virtual array file is named A% in this program. You can assign values to the elements and then use the values.

```
10 OPEN 'ARRAY1' FOR OUTPUT AS FILE #1
20 DIM #1, A%(2000)
```

The next example opens an existing two-dimensional, string virtual array file. The virtual array file is named F$ in this program. Only input is allowed; that is, if you try to assign a value to an element of the array, MINC prints out the following message.

```
?MINC-F-OPEN statement for this file channel prohibits transfer at line 50
```

```
30 OPEN 'ARRAY2' FOR INPUT AS FILE #2
40 DIM #2, F$(100,2)=25
```

In this previous example, each string element can have a maximum of 25 characters. A complete example of a program using a virtual array file is given later in this chapter.

Closing a Virtual Array File

When you open a virtual array file for output you must always close the file in the program with a CLOSE statement. If you do not close the file and the program terminates with an error,

MINC leaves the file in an indeterminate state. That is, some of the changes might be in the file and some might not.

The two general forms of the CLOSE statement are:

CLOSE

CLOSE # expression, # expression, ...

The expression is the channel number of the file to close. You can close more than one file with the CLOSE statement. The CLOSE statement with no arguments closes all open files.

If the program terminates normally (that is, by executing the END statement or the statement with the highest line number), MINC closes all files for you. However, you should put in a CLOSE statement.

Using Virtual Array Files

You use a virtual array file just as you use an array stored in the workspace. For example:

```
10 OPEN 'ARRAY1' AS FILE #1
20 DIM #1, A$(2000)=23
.
.
.
100 A$(5)= 'This is a string'
.
.
.
200 F$=A$(7)
.
.
1000 CLOSE #1
```

In line 100, the program puts the string value 'This is a string' in A$(5). This line stores that value in the file. In line 200, F$ receives the string that is already stored in A$(7). Line 200 inputs a value from the file. Line 1000 closes the file.

Example of Using a Virtual Array File

The following example uses the questionnaires program again. (See pages 101-102.) This time, the array that holds the responses is stored on a volume as a virtual array file. Because the array is now stored on the volume, you can enter some questionnaires now and some later. You no longer lose those questionnaires you typed in when you scratch the workspace.

This time, however, the program should test the input to make

sure that each response is between 1 and 3 (1 = agree, 2 = disagree, 3 = don't care). If the typist enters a value that is not between 1 and 3, and the program does not test the value, the program terminates with an error in statement 40 (the ON/GO TO). If the program terminates with an error, the virtual array file is not closed and its contents are unknown to you (that is, some of the elements might be updated while others might not be).

The new program is as follows:

```
LIST

QUEST                23-MAR-80              09:30:37

5 OPEN 'resp' AS FILE 1
10 DIM #1,R(20,3)
20 P$ = 'Input another questionnaire'
30 GOSUB 150
40 ON Z GO TO 50,100
50 FOR I = 1 TO 20
60 PRINT I; \ INPUT Q
63 IF Q> = 1 THEN IF Q< = 3 GO TO 70
66 PRINT 'input not between 1 and 3' \ GO TO 60
70 R(I,Q) = R(I,Q) + 1
80 NEXT I
90 GO TO 20
100 PRINT ,,'AGREE','DISAGREE',"DON'T CARE"
110 FOR I = 1 TO 20
120 PRINT 'question';I,,R(I,1),R(I,2),R(I,3)
130 NEXT I
140 GO TO 1000
150 REM - subroutine to validate yes or no
160 PRINT P$ \ INPUT S$
170 GOSUB 300
180 IF R$<>'YES' THEN IF R$<>'NO' GO TO 160
190 IF R$ = 'YES' THEN Z = 1
200 IF R$ = 'NO' THEN Z = 2
210 RETURN
300 REM - subroutine to convert lower to upper case
310 U = ASC('a')-ASC('A')
320 R$ = ""
330 FOR K = 1 TO LEN(S$)
340 T$ = SEG$(S$,K,K)
350 IF ASC(T$)> = ASC('a') THEN IF ASC(T$)< = ASC('z') THEN T$ = CHR$(ASC(T$)-U)
360 R$ = R$&T$
370 NEXT K
380 RETURN
1000 CLOSE #1
1010 END

READY
```

Now when you run this program, the questionnaire responses that you type are saved in the file called RESP.DAT. Notice that line 5 does not specify FOR INPUT or FOR OUTPUT. By specifying neither, the program creates the file the first time the program is run and updates the file the rest of the times the program is run.

Note that the IF END statement is not applicable to a virtual array file. Each element is equally accessible. Thus, you do not need to test for the end of the file. You just need to close the file.

DELETING A FILE

Whenever you no longer want a virtual array file or a sequential file on one of your volumes, you can delete the file with the KILL statement.

The form of the KILL statement is:

KILL filespec

The filespec is a string that is a specification of the file. For example:

KILL 'ARRAY1'

If you do not specify the file type, the .DAT file type is the default. If you specify a device (for example, SY1:), you must then specify the entire file specification — including the file type.

The KILL statement is similar to the UNSAVE command. However, the default file type for the KILL statement is .DAT where the default file type for the UNSAVE command is .BAS.

Because you can delete a file from a program with the KILL statement, your programs can open a temporary file to create more room for your program's data, and then can delete the file at the end of the program with a KILL statement.

RENAMING A FILE

You can change the name of a virtual array file or a sequential file with the NAME statement.

The form of the NAME statement is:

NAME file-to-be-renamed TO new-name

The new name must have the same volume specification as the

old one. That is, the NAME statement changes the file name, not the physical location. See the NAME statement in Book 3 for further details. The names are strings and must be enclosed by quotes.

For example:

```
NAME 'PROG1.BAS' TO 'INFILE.BAS'
```

If you do not specify the file type in the NAME command, the default is .DAT.

CHAPTER 12
OTHER BASIC STATEMENTS

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

READ AND DATA STATEMENTS

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.

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

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-

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

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$="#######.## 'RRRRRRRRRRR"
20 PRINT USING A$,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:

"#######.## 'RRRRRRRRRRR"

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.

FORMATTING NUMERIC OUTPUT

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
```

```
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-

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
```

```
***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'
```

```
    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 "'CCCCCC", '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)
100 NEXT I
110 END

READY
RUN

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

```
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.

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 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

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
```

```
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
```

```
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 \ LINPUT #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 \ LINPUT #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:

Segment 1

```
10 COMMON A,B,C$
20 COMMON D(100)
30 COMMON G$(2)
```

Segment 2

```
10 COMMON A,B
30 COMMON C$,D(100),G$(2)
```

Segment 3

```
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.

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

APPENDING PROGRAMS

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'; \ LINPUT F$
```

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.

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.

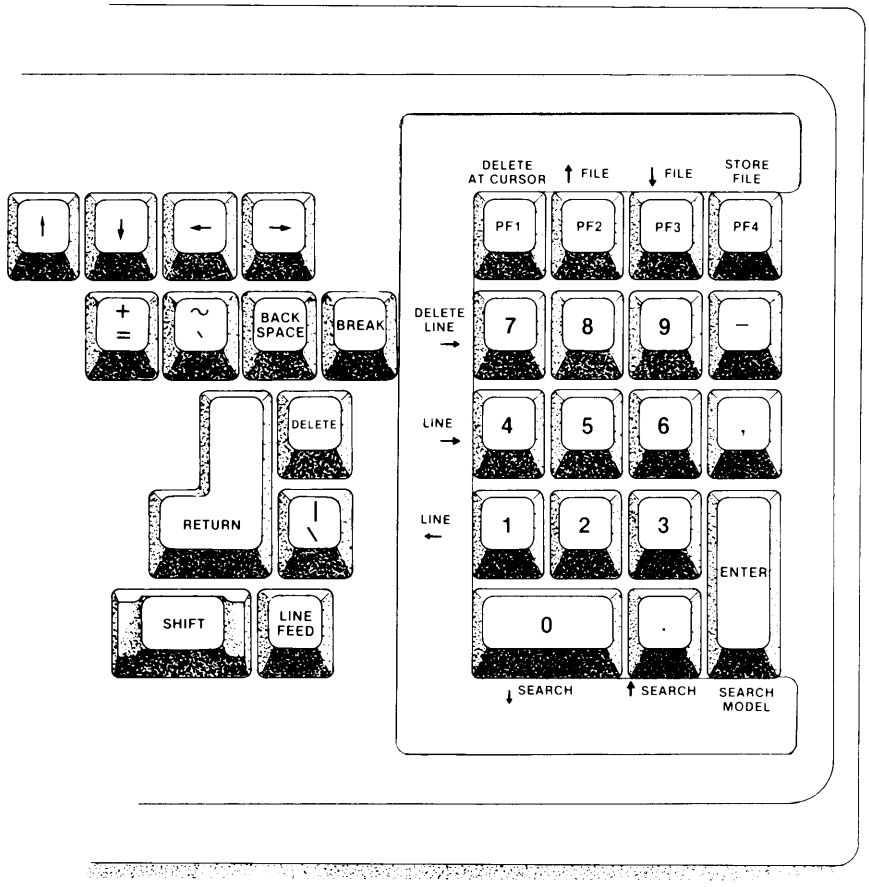


Figure 14. The Keypad

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 additonal operations that are active while you are adding to or modifying the contents of a file. The third section covers the CRE command.

BASIC Programs and the Keypad Editor

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.

- 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.

INSPECTING AN ASCII FILE

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

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

a       SUN    MON    TUE    WED    THU    FRI    SAT
b
c     +------+------+------+------+------+------+------+
d     |  1   |  2   |  3   |  4   |  5   |  6   |  7   |
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
```

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.

Introducing Searching

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.

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 (SP) (SP) (SP) (SP) |, where (SP) 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 | (SP) (SP).

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 (RET) 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 .

EDITING AN ASCII FILE

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

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

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
12345678901234567890123456789012345678901234567890123456789012345678
♦ 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
♦  |15    |16    |17    |18    |19    |
m       |      |      |      *|      |      |      |      |      |      |      |      |...
n       |      |      |       |      |      |      |      |      |
o       +-------+-------+-------+-------+*******+-------+-------+
p       |20    |222   |23    |24    |25    |26    |
q       |       |       |       |       |       |       |
r       |       |       |       |       |       |       |
s       +-------+-------+-------+-------+-------+-------+-------+
+-------+-------+-------+-------+-------+-------+-------+
t       *       |27    |28    |129   |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 | SP *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.

Control Characters and the Keypad Editor

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.

Screen Width and the Keypad Editor

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

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.

CREATING AN ASCII FILE

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.

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 CRTL/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.

ASCII Decimal Code	*Character*
0	NUL (CTRL/@)
1	SOH (CTRL/A)

ASCII Decimal Code	*Character*	
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	,	

ASCII Decimal Code	*Character*
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

ASCII Decimal Code	*Character*
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

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

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1-2/E37
MARLBOROUGH, MASSACHUSETTS 01752

Do Not Tear - Fold Here and Tape