PROGRAM YOUR IBM® PC™ TO PROGRAM ITSELF!

DAVID D. BUSCH

No. 1898
$18.95

PROGRAM YOUR IBM PC TO PROGRAM ITSELF!

DAVID D. BUSCH

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

FIRST EDITION
FIRST PRINTING

Copyright © 1986 by TAB BOOKS Inc.
Printed in the United States of America

Library of Congress Cataloging in Publication Data

Busch, David D.
Program your IBM PC to program itself!

Includes index.
1. IBM Personal Computer—Programming. 2. BASIC (Computer program language) I. Title.
QA76.8.I2594B874 005.265 85-28094
ISBN 0-8306-0898-2
ISBN 0-8306-1898-8 (pbk.)

Contents

Introduction

Are you ready to tap the amazing abilities of one of the most powerful computers on the market? Would you like to create your own DOS commands? Automatically redefine your IBM's special function keys each time you enter BASIC? Command your computer to write programs for you? All that and more is possible using the unique utility programs in this book.

Yes, your IBM PC *can* write its own programs. Instead of laboriously writing program lines and subroutines that will display a series of instructional frames on the screen, you can let your computer do all that work. You need only design the screen, using word processing commands, and tell the computer how long you want that frame displayed. The IBM PC is perfectly capable of writing a complete program that will do exactly that without the need for you to write one single line of code.

Or, your IBM PC can compose subroutines for you. Do you need some disk input/output routines and a string array to store data in? Some data lines, perhaps? A menu? Not too eager to write the code, figure out the proper ON . . GOTO lines? That task is a snap for the automatic IBM PC.

You may be weary of calculating tabs for neatly-formatted screen displays. Wouldn't it be nice to just type PRINT TAB(T) and let the computer figure out what value T should be? Say no more. Your wish is well within the capabilities of the Boca Raton Wonder.

As fabulous a tool as the IBM PC line has been, most users only save half the time they could with their computers. Because I write dozens of programs a year, one of the first things I did was write a number of programs that do nothing more than write other programs for me. One of the first, and one I use more than any other, was Screen Editor. It is a BASIC program that allows drawing on the screen any menu, title block, instructional screen, or other material that will be needed in a program. Then, at the press of the Enter key, the screen just designed is magically transformed into program lines. Ten minutes of coding can be accomplished in a minute or less. (Actually, since I have compiled into machine code the BASIC Screen Editor shown in this book, the chore takes no more than a second or two!)

I also let my Automatic PC use DOS commands that I have created. I don't have to type DIR B: and then watch as the directory flies past on the screen. Instead, I type D B, and see a paged listing that pauses until I am ready to continue. When I want to examine a file from DOS, I just type LOOK file name B and my PC displays the file, again a screenful at a time. This book shows you how to install your own favorite DOS commands and customize your computer with a special prompt and other features.

Given the right tools, such as the utility programs here, an hour spent programming can be more fruitful than several hours under manual methods. A third or more of the program lines in some of the examples in this book were prepared by other programs listed. Some programs were even used to write enhanced versions of themselves.

All the programs in this book will work with IBM PC and PCjr computers with 80-column displays, Most will also operate on the PCjr with a 40-column display, with a few changes. Tabber, for example, asks the user if tabs should be centered for a 40-column screen, or an 80-column screen. Because they are written in BASIC and use no PEEKs, the programs are readily transferable.

Just as PEEKs and POKEs have been avoided wherever possible, other statements that are DOS dependent have been avoided. In most cases, strictly BASIC syntax common to all IBM PC computers is used. All versions of PC-DOS have similar disk input/output routines for sequential files, which are used in most of the programs in this book.

This book is only a jumping off point. Many of the programs were adapted from other programs. Visual Maker is based on Screen Editor. Global is descended from Tabber. Similarly, you can take the ideas and suggestions here and develop programs of your own that will streamline your BASIC development work. In addition, there are some ideas in Chapter 16 for using other programs you already own—such as word processors or keyboard utilities—as shortcuts.

The utility programs in this book actually write programs for you, modify existing software, or give your programs new capabilities and power. Hours of time can be saved on every program written by the novice or experienced programmer. Some of the examples were used to write programs in this book or to modify themselves.

Here is a brief outline of the programs:

Visual Maker. Design a custom "slide" to appear on the screen of your IBM PC, using graphics or alpha characters. Tell Visual Maker how long you want that slide to be displayed. Then go on to the next slide.

Once assembled into the order you want, Visual Maker will write a complete BASIC program to display the slides you designed for the intervals requested. No programming is required.

DB Starter. Weary of writing custom database management programs from scratch? DB Starter will do the BASIC skeleton for you. Enter the number of menu choices and the prompts to be included in the menus. It will design the menu for you. Tell the program you want Input/Output routines and feed in a few parameters; it will write the I/O modules automatically. DB Starter will also construct the necessary ON . . GOTO lines and insert REMarks at line numbers where the programmer needs to build up the BASIC skeleton. Your first several hours of programming are taken care of for you.

Tabber. Want to enter your screen output for prompts and other messages? Just type PRINT TAB(T) in every line you want centered. This program will go through an entire program, calculate how long the message is, and write a new program line that TABs the proper number of spaces. Works with 40- or 80-column screens.

Proofer. Find misspelled keywords, mismatched parentheses, and other errors BEFORE runtime. This program helps you debug and provides a list of variable names used in the program as a bonus.

Error Message. If you are impressed with the long error messages of BASICA, this program will knock you flat. Append Error Message to your own

BASIC program and insert the appropriate ON ERROR GOTO . . line. Then, any error will be spelled out in detail—with tips on how to find the exact error in your program. This will prove to be an excellent utility for novice programmers or anyone tracking an elusive bug.

Other programs in the book include:

Screen Writer. Use word processing-like commands to design a custom screen. Then, press the Enter key. This program writes the BASIC program lines you need to reproduce your custom screen in your own program.

Key Definer. You enter the function key definitions you want. This program writes a simple BASIC program that will make the changes for you automatically and then erase itself. The file is stored on your disk so you can invoke your new function key definitions automatically when you enter BASIC—or choose from several sets of definitions!

Word Counter. Count the words in your document or program. This program works with any ASCII file.

Global Replacer. Specify a string in your program—it does not have to be a keyword—and this program replaces it with a string of your choice.

REM-Over. Take remarks out of your program automatically.

Lister. Format your BASIC programs for listing on your printer.

Translator. This program allows you to write programs in a foreign language, such as French or Spanish. It then translates them to standard IBM PC BASIC for running.

How to Use This Book

All the programs in this book have been rigorously tested and will run as described. Working program listings were transferred directly to a word processing program where REMarks were added; the listings were printed out with no further changes. Those printouts were photographed for this book.

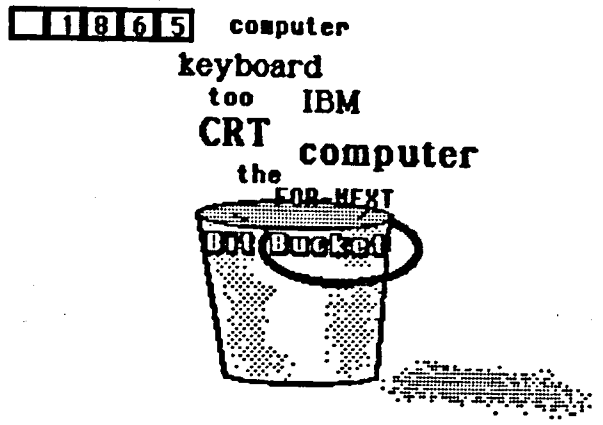
So, if you type in a program and have difficulty making it work, odds are very good that a small typing error could be the source of your problem. Go back and proof each line carefully. Up to 40 characters are significant in IBM BASIC, so a variable name that is typed in as TOTAL in one place and TOTALS in another are two completely different variables.

These programs have been written to make your job as easy as possible. Variable names have been chosen to be descriptive without excessive length. That is, the variable name CHAR might be used to count the number of characters in a document instead of the variable name CHARACTERS.IN.DOCUMENT.

Some magazines tightly pack program listings to save space. Most program lines here have only one or two statements, except where IF . . THEN . . ELSE logic dictates more complex construction. I've tried to indent FOR-NEXT loops and mark each module within a program with a REMARK so you can see what happens where. Some consistency is used in variable names between programs as well. F$ is most often used for disk file names, A$ for INKEY$ and other INPUT uses, T$ or TEMP$ for temporary string variables, and so forth. Once you've learned the conventions of this book, it will be easier to follow the program logic.

If you like, you can even use some of the programs in this book to reduce your work. You may, for example, choose to abbreviate some frequently used statements, and then run the Global search and replace program to make a substitution. Instead of typing PRINT TAB(dozens of times, abbreviate it with PZ. You will not be able to debug the program until the change is made. Once you have typed all the program lines in, however, save what you have in ASCII format (more on that later) and ings and mismatched parentheses. When the basic work has been done for you, then, and only then,

Remember the next time you are slaving over a hot keyboard at 3 A.M. that computers are the servants of mankind, not vice-versa.



Word Counter

Why not let your IBM PC write its own programs? After all, much of program writing is nothing more complicated than building something from an inventory of prefabricated subroutines. Many programs have a great deal in common; it is the parameters that change. Wouldn't it be simpler just to provide the parameters and let the computer do the routine stuff?

BUILDING A LIBRARY OF ROUTINES

One program may require a line like FOR N=1 TO 100, while the next will need FOR N=1 TO 200. Yet, each time, the programmer had to type in the FOR N=1 TO part. The reason the computer never knew enough to supply the FOR N=1 TO is that nobody told it to. The IBM PC and PCjr computers can do practically anything in the area of program writing, if they are only told exactly what to do.

Applications generators and other programs that write other programs are old hat. They have been around for a number of years and can be purchased for large computers as well as small. The concept behind them is simple: many programs have modules that are much like those used in other software. Yet, in many cases the computer programmer writes a routine from scratch each time it is needed. Why not build a library of routines and let the computer draw on them as needed to write its own programs?

The basis behind why an IBM PC can write its own BASIC programs lies in its ability to load into BASIC from disk two types of files. The normal way a BASIC program is saved is in compressed format. That is, BASIC keywords are tokenized, and a single byte representing that keyword is loaded onto the disk, instead of the entire keyword. Rather than store the five letters that make up "PRINT," BASIC normally just stores a one-byte decimal number that represents "PRINT." When you type SAVE"file name.bas", a program is stored on disk in this form.

You can also, however, type SAVE"file name.bas",A. Then the program will be saved in noncompressed *ASCII format*. That is, every letter and number will be stored, byte for byte, on the disk exactly as the program appears when listed. The BASIC interpreter has the capability of doing this conversion for us. An ASCII file is nothing more than a text file. It is possible to load a noncompressed program into a word processing program, edit it using powerful global search and replace commands, and then save it back to disk in ASCII form. Some word processors do not normally save in ASCII format, but most have an option or utility that allows you to do this.

Because of BASIC's dual capability, you can also create programs using a word processor or, in the case of the programs in this book, through the use of sequential disk files, which are also ASCII files. The short program below serves as an example:

```
10 OPEN "O",1,"TEST.BAS" (or,
   OPEN "TEST.BAS" FOR OUTPUT
   AS 1)
20 PRINT #1,"10 PRINT";CHR$(34);
   "THIS IS A TEST";"CHR$(34)
30 CLOSE 1
```

This test program will write a single line to the disk under the file name TEST.BAS. That line will be, if loaded into BASIC, a short program in the form:

```
10 PRINT"THIS IS A TEST"
```

You could also "build" the program lines from your own parameters. Try this short program:

```
10 INPUT"Enter line number
   desired:";LN
20 INPUT"Enter message desired
   :";MESS$
30 INPUT"Want it to be PRINT or
   LPRINT";CH$
40 IF CH$="PRINT" OR CH$=
   "LPRINT" GOTO 60
50 GOTO 30
60 OPEN "O",1,"TEST.BAS"
70 PROG$=STR$(LN)+CHR$(32)+CH$
   +CHR$(32)+CHR$(34)+ MESS$
   +CHR$(34)
80 PRINT #1,PROG$
90 CLOSE 1
```

Most of the programs in this book with program writing routines do nothing more than assemble program lines in this manner. Sometimes the input comes from the user. Other times it is calculated. Still other times, some of the programs use the SCREEN function to see what has been printed to the screen and use that information.

HOW WORD COUNTER WORKS

The common thread among the programs is the use of ASCII files that are programs as if they were data files. The first program presented, Word Counter, illustrates the principle, even though it does not create any new program files itself. Instead, Word Counter reads in an ASCII file and counts the number of words. Most commonly, these files will be word processing text files. Word Counter, however, could just as easily be used to count the number of words in a program.

Most of the techniques used in this book will be repeated in later programs. Each will be explained in detail the first time they are used. So, early programs are short because explanations are frequent. Later, longer programs will use many techniques that have been previously explained and will thus require fewer discussions.

Programs in this book frequently access other programs that have been stored in ASCII form on disk. You *must* save a program to be used by another program in ASCII form using the ,A option. If, in running one of the programs here, you see garbage on the screen, you probably forgot to save the program in ASCII.

Word Counter is no exception. It will count words in a program file just as it will the words in a text file, but only if both are in ASCII. Figure 1-2 presents the variables used in Word Counter. The operator is asked to enter the name of the file to

A$	Stores text line being examined.
AW	Average word length in text.
C$	One-character string from middle of line.
CHAR	Number of characters in whole file.
CU	Counter of number of words in file.
F$	Text file to be counted.
FL	FLAG indicating end of file reached.
L$	Last character encountered.
N	Loop counter.
SW	Number of standard words in text.

Fig. 1-1. Variables used in Word Counter.

be processed in line 280. That file, F$, is opened, and one line is input from the disk. The line is loaded by means of LINE INPUT #1 in line 390. INPUT #1 will accomplish much the same thing, except that it will not accept string delimiters, such as commas and quotation marks, which are commonly used in both text and program lines. LINE INPUT imposes no such restriction. It accepts everything up to the next carriage return. This will be the end of a program line or a carriage return in the text itself.

To search for a word, you need to first decide just what a word is. The easiest thing is to realize that a word is more or less a group of letters preceded and followed by a space. "CODEWORD" is one word, even though two real words are embedded in it. "OH! NO!" is two words. The punctuation is not part of each word, but for the purposes of this program, it is considered so. This is because Word Counter has been written to look for each space that is preceded by a nonspace. Counting spaces would be an inaccurate way of counting words. There would probably be two spaces following a sentence, for example. So, the program instead looks at each character; when it finds a space, it looks to see if the last character was a space. If not, the end of a word has been deemed to have been reached.

Each line input, stored in A$, is looked at one character at a time in a FOR-NEXT loop beginning at line 420. The loop repeats from 1 to the length of A$. Each time through, C$ is assigned the value of the next character in the string, through the use of MID$(A$,N,1). MID$, as you know, takes the middle portion of a string, starting at position N (in this case) and with a length that you specify. In the above example, just one character was selected.

If C$ is a space, (CHR$(32)), the program looks at the last character checked, L$, to see if it was a space. If it was *not* a space (that is, it was a character), the program assumes that the end of a word has been found, since no word contains an embedded space. Thus, the word counter, CU, is incremented by one.

Before the loop goes back to look at the next character, the current character is stored in L$ (line 380) and becomes the last character.

Once the program has looked at every character in the string, it drops down to line 470 where the end-of-file flag is tested. If it is one, meaning the EOF marker has been reached, the program goes to line 490 to present the results of the word count. Otherwise, the program goes back to line 390 to input another line.

When the file is finished, the program prints the number of words, CU, and then calculates the average word length, which is the number of characters (CHAR) divided by the number of words. The number of characters is also divided by five to total the amount of "standard," five-character words as well. Of course, most words will be longer or shorter than five characters, but I use this length as an average to determine how many words are in a document.

THE INTERRUPT ROUTINE

In nearly all of the programs in this book, you may abort at any time by pressing the F10 function key. This is set up in line 90 of this program, with the ON KEY(10) command. This is an *interrupt-driven* routine, meaning that the PC will execute the specified subroutine at any time (almost) that it is triggered. The program does not have to be sent to the subroutine by encountering an ON . . GOSUB line. Instead, you can turn the feature "on" or "off" as you wish. This is done in line 100. While ON KEY is on, pressing F10 will send control to line 660, where all files are closed and the program is ended. If you want to turn the feature off temporarily, a KEY (10) OFF statement could be placed in the program.

Note that if the program is waiting for certain types of input, such as through the use of INPUT$, the interrupt routine will not be triggered until after you make the expected input. For example, here are two ways of pausing until the user presses a key:

```
100 A$=INPUT$(1)
110 A$=INKEY$:IF A$="" GOTO 110
```

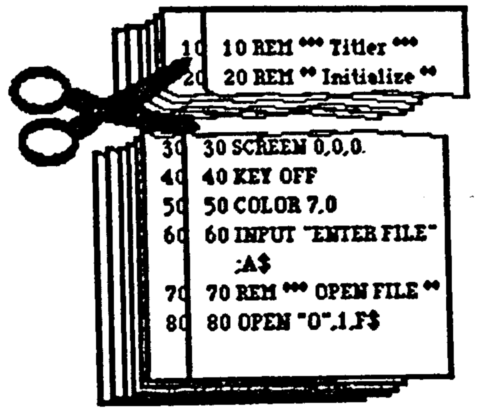
The former is considered by some to be more elegant; however, if you press F10 and no other key, the program will wait forever at line 100. With line 110, though, the IBM PC will immediately proceed to the interrupt subroutine whenever F10 is pressed. For that reason, I've generally stuck to the use of INKEY$ in this book.

Listing 1: The Word Counter Program

```
10 ' ********************
20 ' *                  *
30 ' *    Word Counter  *
40 ' *                  *
50 ' ********************
60 DEFINT A-Z

65 ' *** Instructions ***

70 KEY OFF
80 SCREEN 0,0,0
90 ON KEY(10) GOSUB 660
100 KEY (10) ON
110 CLS:PRINT:PRINT
120 COLOR 0,7
130 LOCATE 4,24
140 PRINT " Writer's Word Counter "
150 COLOR 7,0
160 PRINT
170 PRINT TAB(14)"This program will count the number of
    actual words in a "
180 PRINT TAB(10)"text file, or any file that has been
    stored to disk in ASCII
190 PRINT TAB(10)"format.  In addition, it also provides the
    total number of "
200 PRINT TAB(10)"'standard ' five-character words, and the
    average character "
210 PRINT TAB(10)"length of the words in the text. "
220 PRINT:PRINT TAB(22)"";
230 COLOR 0,7
240 PRINT " == Hit any key to continue == "
250 COLOR 7,0
260 IF INKEY$="" GOTO 260
270 CLS:PRINT:PRINT

275 ' *** Access Disk File ***

280 PRINT TAB(15)"Enter name of file to count: ";
290 LINE INPUT F$
300 CLS
310 LOCATE 15,25
320 COLOR 16,7
330 PRINT"Counting file ";F$
340 COLOR 0,7
350 LOCATE 25,27
360 PRINT" Hit F10 To Abort ";
370 COLOR 7,0
380 OPEN "I",1,F$
390 LINE INPUT #1,A$

395 ' *** If End of File Found, Set Flag to 1 ***

400 IF EOF(1) THEN FL=1

405 ' *** Add Length of A$ to Total Characters in File ***

410 CHAR=CHAR+LEN(A$)

415 ' *** Loop to look at each character in A$ ***

420 FOR N=1 TO LEN(A$)
430 :       C$=MID$(A$,N,1)
440 :       IF C$=CHR$(32) AND L$<>CHR$(32) THEN CU=CU+1
450 :       L$=C$
460 NEXT N
470 IF FL=1 GOTO 490
480 GOTO 390
```

```
485 ' *** Print out Results ***

490 CLS:PRINT:PRINT
500 PRINT TAB(23)"NUMBER OF WORDS =",CU
510 PRINT
520 AW=CHAR/CU
530 PRINT TAB(21)"AVERAGE WORD LENGTH =",AW
540 PRINT
550 SW=CHAR/5
560 PRINT TAB(17)"NO. OF FIVE-CHARACTER WORDS =",SW
570 CLOSE

575 ' *** Run again? ***

580 PRINT:PRINT
590 PRINT TAB(22)"Check another file?"
600 LOCATE 15,30
610 COLOR 16,7
620 PRINT" Y/N ?"
630 COLOR 7,0
640 A$=INKEY$:IF A$="" GOTO 640
650 IF A$="Y" OR A$="y" THEN RUN
660 CLOSE
670 CLS
680 END
```

Chapter 2



REM-over

In Chapter 1 we explored opening an ASCII disk file, either text or program, reading it in line by line, and then examining the string of characters in order to count the number of words. The next step is to alter the file in some way and then write a new, changed file to disk. Several of the programs in this book are based on that principle. The first of these is "REM-over."

THE PURPOSE OF REM-OVER

The program will read in a disk file, like before. REM-over, however, will print to disk a new file that is similar to the old one. The only difference is that when the program encounters a remark, designated either by "REM" or its abbreviation "'", the remainder of the program line will be truncated. If a line consists only of a line number and a remark, the line will be deleted from the program entirely. The result will be a new program with all of the comments removed. Depending on the number of remarks included in the original program, the new, remarkless version may be significantly smaller, and therefore consume less memory space. Figure 2-1 shows an example of a program that contains remarks; Fig. 2-2 shows this program after REM-over has been used with it.

HOW REM-OVER WORKS

Ordinarily you might think that deremarking a program would be ridiculously simple. Since the IBM PC ignores anything after REM or ', a program could simply search for those two strings. You should, however, realize that REM or ' within quotation marks doesn't "count." That is, when REM is used as part of an input prompt or in a PRINT statement, it does NOT appear to be a remark to the computer. For example:

```
10 PRINT"This is NOT a REMark."
   :REM But this IS.
```

REM-over takes care of this stipulation by simply looking at each program line for quotation marks as well as remarks. If a REM appears after,

```
10 ' Test of Program REM-OVER
20 REM Will Test REMOVAL of REMARKS
30 ' This Remark will be removed.
40 PRINT:PRINT: REM This one will be removed.
50 PRINT"This REMARK: REM Will NOT be removed."
60 PRINT"This one won't":REM This one will.
```

Fig. 2-1. Target program for REM-over.

one quote but before the second, it is contained within the quotation marks. This assumes that the programmer has not mismatched quotation marks and has included two for every prompt. In fact, the program will "crash" if it encounters a line like this:

```
10 PRINT "This is NOT a REMark.
```

Notice that the second quotation mark is missing? BASIC will run this line just fine, even without the quotation mark, but the omission is not good programming practice. It will cause REM-over to hiccup rather badly.

Figure 2-3 provides the variables used in REM-over. The program begins by asking the operator for the file name of the program that will have its remarks REM-oved. This file name, F$, is used to form the file name of the output file automatically. In line 150, the second file name, F1$, is formed by adding ".REM" onto it. If the file name happens not to have an extension, as, for example, when F$="TEST", the new file name, "TEST.REM," will be legal. Of course, BASIC program names must end in .BAS, but you can change these DOS mode using the RENAME TEST.REM TEST.BAS syntax.

A check is made later in line 150 to see if the original program name includes a period and an extension. F1 is equal to INSTR(F$,"."). If F1=0, that is if F$ does *not* contain a period and extension, the program goes to line 160.

If however, a period *is* found, and F1 does NOT equal zero, the portion of the file name up to the period (".") (LEFT$(F$,F1-1)) is taken, and ".REM" is tacked on. Next, both files are opened, and a single line is input in line 250. Variable P, which is the position at which the search for REMs begins, is set to one. Thus, the initial search for remarks will begin at the first character of A$.

Because both REM and ' can indicate remarks, two searches must be conducted. First, in line 270, the program checks for ' and, if an apostrophe is found, assigns the position of the suspected remark to the variable R. Control then branches to line 310.

If no apostrophe is located, the program next checks for "REM", in line 290. If no remark is found, then the program line is already remark-free, and the program branches to line 460.

Possible remark lines are examined further at a routine beginning at line 310. There, Q1 is assigned the value of the position of a quotation mark. If none is found, then a remark has indeed been located and the control passes to line 370. If a quotation mark is detected, then REM-over looks

```
40 PRINT:PRINT
50 PRINT"This REMARK: REM Will NOT be removed."
60 PRINT"This one won't"
```

Fig. 2-2. Example of a program with REM-marks removed.

A$	Line of program loaded from disk.
B$	Middle string of program line.
F$	File name of program being processed.
F1$	File name of output file.
N	Loop counter.
P	Position to begin INSTR search.
Q1	Position of first quote mark.
Q2	Position of second quote mark.
R	Position of remark.
T$	String remaining after remark deleted.

Fig. 2-3. Variables used in REM-over.

at the rest of the program line, beginning at position Q1+1 for a second quotation mark. That value is Q2. If the position of the remark, R, is less than Q1 (the remark appears *before* the first quotation mark) or is more than Q2 (it appears *after* the second quotation mark, then a remark is verified, and the program goes to line 370.

If neither condition is true, the alleged remark is actually within the quotation marks and is disqualified. The program instead makes P equal to the next position after the second quotation mark (Q2+1) and returns control to line 270 to see if any possible remarks exist after position P. In this way, an entire, multistatement line can be looked at section by section to detect all remarks.

When a valid remark is located, the program takes all of the program line up to the remark itself, using A$=(LEFT$,R-1), as in line 370. This, in effect, truncates the program at the remark.

We are not through yet. After all, some program lines consist of just a line number and a remark. Cutting off the remark leaves only the line number. This is a bit untidy and a waste of computer memory as well. So, the program cycles through a FOR-NEXT loop from 1 to the length of A$. Each time through, the string variable B$ is assigned the value of the middle character at position N. This character is checked to see that it is a number in the range 0-9, since all program lines begin with numbers. As soon as B$ does *not* equal a number, REM-over knows that the line number is over, and control drops down to line 420.

There, T$ is assigned the rest of A$. IF T$ is empty, or consists only of a space, the program knows it has found an "empty" program line and loops back to line 230 without printing anything to the disk. That line has been deleted from the program entirely.

There, T$ is assigned the rest of A$. If T$ is see if the final character is a colon, as would be the case if a remark followed a colon on a multistatement line:

```
10 PRINT"HELLO":REM This is a
   remark.
```

If a colon is the last character, it is deleted in line 450. A$ is printed to the screen, so the operator can monitor the progress of the program, and printed to the disk. Control goes back to line 230, where a check for the end-of-file is made and another program line input from the disk.

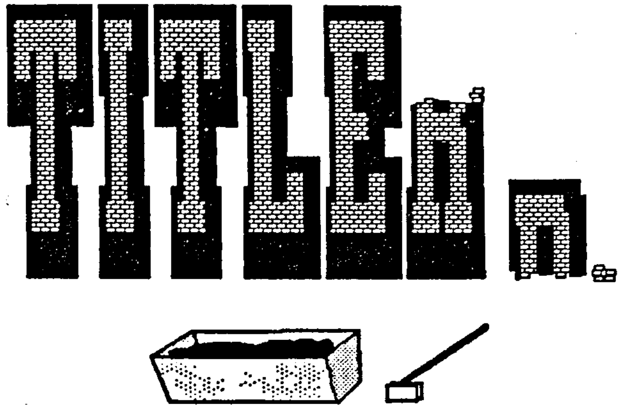
That's all there is to REM-oving the REM-arks from your IBM PC programs.

Listing 2: The REM-over Program

```
10 ' ****************
20 ' *              *
30 ' *   REM-over   *
40 ' *              *
50 ' ****************

55 ' *** Initialize ***

60 SCREEN 0,0,0
70 KEY OFF
80 ON KEY(10) GOSUB 550
90 ON ERROR GOTO 580
100 KEY(10) ON
110 COLOR 7,0

115 ' *** Enter filename ***

120 CLS:PRINT:PRINT
130 PRINT TAB(17)"Enter name of program to have REMARKS
    removed:"
140 LINE INPUT F$
150 F1$=F$+".REM":F1=INSTR(F$,"."):IF F1=0 THEN GOTO 160 ELSE
    F1$=LEFT$(F$,F1-1)+".REM":GOTO 160
160 OPEN "I",1,F$
170 OPEN "O",2,F1$
180 CLS
190 LOCATE 25,30
200 COLOR 16,7
210 PRINT" Hit F10 to abort ";
220 COLOR 7,0
230 LOCATE 10,1
240 IF EOF(1) GOTO 490

245 ' *** Load Program Line ***

250 LINE INPUT #1,A$
260 P=1

265 ' *** Check for REMARKS ***

270 R=INSTR(P,A$,"'")
280 IF R<>0 GOTO 310
290 R=INSTR(P,A$,"REM")
300 IF R=0 GOTO 460
```

```
305 ' *** Find Quotes, if Any ***

310 Q1=INSTR(P,A$,CHR$(34)):IF Q1=0 GOTO 370
320 Q1=Q1+1
330 Q2=INSTR(Q1,A$,CHR$(34))
340 IF R<Q1 OR R>Q2 GOTO 370
350 P=Q2+1
360 GOTO 270

365 ' *** Strip off REMARKS ***

370 A$=LEFT$(A$,R-1)
380 FOR N=1 TO LEN(A$)
390 B$=MID$(A$,N,1)
400 IF ASC(B$)<48 OR ASC(B$)>57 GOTO 420
410 NEXT N
420 T$=MID$(A$,N)
430 IF T$="" GOTO 240
440 IF T$=" " GOTO 240
450 IF RIGHT$(A$,1)=":" THEN A$=LEFT$(A$,(LEN(A$)-1))

455 ' *** If line not empty, print to disk ***

460 PRINT A$
470 PRINT #2,A$
480 GOTO 240
490 CLOSE

495 ' *** Again? ***

500 PRINT:PRINT
510 PRINT TAB(29)"Process another file?"
520 PRINT TAB(37)"(Y/N)"
530 A$=INKEY$:IF A$="" GOTO 530
540 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
550 CLOSE
560 CLS
570 END

575 ' *** Error Routine ***

580 IF ERR=53 THEN PRINT"That file is not on your disk! Try
    again.":PRINT:PRINT:RESUME 130
590 PRINT"Unknown error in line "ERL;".":RESUME 130
```

Chapter 3



Titler

Now you are ready for some real action. Making a few simple changes in an existing program is kid stuff compared with the "real" thing—that is, generating a new, never-before-existing program line from your very own parameters. That's the function of Titler. This program generates program title blocks, such as the one shown in Fig. 3-1, that can be merged with your own programs. You don't have to tediously write the program lines yourself, format the title block, or even supply your name and address every time. The program will do that for you. As an added feature, your friends can also use the program by supplying their own names.

ENTERING DIFFERENT NAME AND ADDRESS TITLES

This capability is carried out through what are known as *default* values. That is, the programmer assigns values to the name, address, and city variables. (See Fig. 3-2 for a complete list of variables used in the program.) Every time the program is run, the user can simply press the Enter key when asked whether or not a new name and address should be input. The question is posed in line 220. Then an INKEY$ loop repeats until the operator presses a key, or presses the Enter key. If N or the Enter key (CHR$(13)) was pressed, the program drops down to line 400, and N$, AD$, and CT$ remain as they were defined in lines 60-80. The default values are used.

If Y or some other key is pressed, however, the program will ask for a name, address, city, state, and zip, and will assemble the string variables N$, AD$, and CT$ on its own. In that way, a regular user can be accommodated, while a path is left open for a friend to use the program as well.

INCLUDING TIME IN THE TITLE

If you want, information other than the name or address can be incorporated into the title. You might add a line:

```
325  CT$=TIME$
```

and delete lines 330 through 390.

```
1 ' **************************
2 ' *                        *
3 ' *      Title Maker       *
4 ' *                        *
5 ' *     David D. Busch     *
6 ' *  515 E. Highland Ave.  *
7 ' *   Ravenna,Ohio 44266   *
8 ' *                        *
9 ' **************************
```

Fig. 3-1. Sample title produced by Titler.

In this case, the time when the title was created will be embedded in the title block instead of the city and state names. If you keep your system clock accurate, this can be a good way to keep track of various versions of the same program!

GENERATING PROGRAM TITLE BLOCKS

Next, the user is asked for the title of the program he is going to add the title block to; this is stored in TITLE$. The program checks to see which of the four strings—the program title, the name, the address, or the city—is the longest. The longest of the four strings determines how wide the title block will be. This width, A, is defined in line 470 as the length of the longest string plus 4. The extra four characters will leave room for a space at each end of the longest string plus an asterisk used as the border.

A disk file named TITLE.BAS is opened, and a subroutine at line 890 is accessed to produce a string that contains the next line number that will be used in our miniprogram. What this subroutine does is increment a counter, LC, each time it is called. Then, LN$ is formed by converting the counter LC to a string value and adding an apostrophe, because our title block will consist of remarks, and a space, CHR$(32). Then the subroutine RETURNs to the main program.

There LN$ is added to the beginning of a string equal to A+2 in length, consisting of all asterisks. So, the first line might look something like this:

```
1 ' ********************
```

That line is PRINTed to the disk in line 520. Then, the subroutine at 890 is called again, and a new line is formed similarly. This line consists of a line number that is one greater than the last, the apostrophe, an asterisk, followed by spaces equal to A, and another asterisk. This line will look like this:

```
2 ' *                    *
```

The following line will contain the title itself and will have an asterisk, some spaces, the title, some more spaces, and another asterisk. The number of spaces fore and aft will be divided as equally as possible at each end, so that the title will be centered. These are calculated by subtracting the length of the title from A, dividing that by 2, and assigning that value to the number of spaces preceding the title, B1. The number of spaces following is the number remaining after subtracting B1 from B. This is done, instead of simply dividing B by two, because the result will not always be even. It is sometimes necessary to make B1 one space larger than B.

This centering procedure is repeated in the following lines in which the name, address, and city are included in the title block. The block is finished when a program line identical to line 1 is written to the disk.

The last step is to close the file and print instructions to the user that tells him to renumber the target program so that the first line number is higher than 10 and then MERGE his program with the TITLE file.

There, we have created a program from nothing. Next, things get a little more complicated.

A$	Using in INKEY$ loop.
A	Length of widest line in title.
AD$	User's address.
B	Difference between length of line to be incorporated in title and A.
B1	Number of spaces before line.
B2	Number of spaces after line.
C$	User city.
CT$	Name of user's city, state, zip.
LC	Line counter.
LN$	Program line currently being built.
N$	User's name.
N	Loop counter.
S$	User's state.
TITLE$	Title of program.
Z$	User's Zip code.

Fig. 3-2. Variables used by Titler.

Listing 3: The Titler Program

```
10 ' ********************
20 ' *                  *
30 ' *  Program Titler  *
40 ' *                  *
50 ' ********************

55 ' *** Defaults ***

60 N$="Your Name Here"
70 AD$="Your Address Here"
80 CT$="Your City, State, Zip"
90 KEY OFF
100 SCREEN 0,0,0
110 COLOR 7,0
120 ON KEY(10) GOSUB 910
130 KEY(10) ON
140 CLS:PRINT
150 LOCATE 25,29
160 COLOR 16,7
170 PRINT" Hit F10 to abort ";
180 COLOR 0,7
190 LOCATE 8,31
200 PRINT "Title Block Writer"
210 COLOR 7,0
220 PRINT:PRINT TAB(28)"Enter Name and Address?"
```

```
230 PRINT:PRINT TAB(36)"Y/N ?"
240 PRINT
250 PRINT TAB(23)"(Just Hit <ENTER> to use Defaults)"
260 A$=INKEY$:IF A$=""GOTO 260
270 IF A$=CHR$(13) OR A$="N" OR A$="n" GOTO 400

275 ' *** Enter Name, etc. ***

280 CLS:PRINT
290 PRINT TAB(34)"Enter name :  ";
300 INPUT N$
310 PRINT TAB(32)"Enter Address :  ";
320 INPUT AD$
330 PRINT TAB(32)"Enter City :  ";
340 INPUT C$
350 PRINT TAB(33)"Enter State :  ";
360 INPUT S$
370 PRINT TAB(32)"Enter Zip Code :  ";
380 INPUT Z$
390 CT$=C$+","+S$+" "+Z$
400 CLS:PRINT
410 PRINT TAB(28)"Enter title of program :  ";
420 INPUT TITLE$
430 A=LEN(TITLE$)
440 IF LEN(N$)>A THEN A=LEN(N$)
450 IF LEN(AD$)>A THEN A=LEN(AD$)
460 IF LEN(CT$)>A THEN A=LEN(CT$)
470 A=A+4

475 ' *** Open Disk file ***

480 OPEN "O",1,"TITLE.BAS"
490 CLS
500 GOSUB 890
510 LN$=LN$+STRING$(A+2,"*")
520 PRINT #1,LN$
530 GOSUB 890
540 LN$=LN$+"*"+STRING$(A,32)+"*"
550 PRINT #1,LN$
560 GOSUB 890
570 B=A-LEN(TITLE$):B1=INT(B/2):B2=B-B1
580 LN$=LN$+"*"+STRING$(B1,32)+TITLE$+STRING$(B2,32)+"*"
590 PRINT #1,LN$
600 GOSUB 890
610 LN$=LN$+"*"+STRING$(A,32)+"*"
620 PRINT #1,LN$
630 GOSUB 890
```

```
640 B=A-LEN(N$):B1=INT(B/2):B2=B-B1
650 LN$=LN$+"*"+STRING$(B1,32)+N$+STRING$(B2,32)+"*"
660 PRINT#1,LN$
670 GOSUB 890
680 B=A-LEN(AD$):B1=INT(B/2):B2=B-B1
690 LN$=LN$+"*"+STRING$(B1,32)+AD$+STRING$(B2,32)+"*"
700 PRINT #1,LN$
710 GOSUB 890
720 B=A-LEN(CT$):B1=INT(B/2):B2=B-B1
730 LN$=LN$+"*"+STRING$(B1,32)+CT$+STRING$(B2,32)+"*"
740 PRINT #1,LN$
750 GOSUB 890
760 LN$=LN$+"*"+STRING$(A,32)+"*"
770 PRINT #1,LN$
780 GOSUB 890
790 LN$=LN$+STRING$(A+2,"*")
800 PRINT #1,LN$
810 CLOSE

815 ' *** Final Instructions ***

820 CLS:PRINT
830 PRINT TAB(18)"Renumber your target program so that
    first"
840 PRINT TAB(18)"line number is higher than 10, then type"
850 PRINT
860 PRINT TAB(29)"MERGE ";CHR$(34);"TITLE.BAS";CHR$(34)
870 PRINT
880 END

885 ' *** Increment Line numbers ***

890 LC=LC+1:LN$=STR$(LC)+"'"+CHR$(32)
900 RETURN
910 CLOSE
920 CLS
930 END
```

Chapter 4

```
10 PRINT TAB(T) "This program will insert"
20 PRINT "tabs into your program lines"
30 PRINT "to make for a much more"
40 PRINT "attractive screen appearance"
```

Tabber

Time for a breather. Tabber is a simple yet elegant little program that will be very useful to you. It creates no new program lines, doesn't make your computer operate 50 percent faster, and won't even make your laundry whiter.

ADVANTAGES OF TABBER

What Tabber will do is automatically center various prompts that are printed to the screen using PRINT or INPUT statements. Instead of sloppy screen formatting, you can have neat copy. It will work with both 40- and 80-column screens of PCs or PCjrs. Best of all, you need to make only one small change in your programming habits.

To center any prompt, simply type PRINT TAB(T) instead of calculating the proper tab position yourself. With messages that were going to be PRINTed to the screen, just insert TAB(T), as shown in Fig. 4-1. If a program presently includes the prompt after an INPUT or LINE INPUT statement, you will have to do some rewriting, since there is no such thing in IBM BASIC (yet) as an INPUT TAB(n) or LINE INPUT TAB(n) statement.

Use the second line rather than the first in the examples below:

```
WRONG: 10 INPUT "Enter your
          name:";A$

RIGHT: 10 PRINT TAB(T)"Enter your
          name:";:INPUT A$
```

You can still run or test programs using TAB(T) before they have been run through Tabber. This is especially useful during program development and testing. Simply insert TAB(T) as you go along. Until the finished program has been processed by Tabber, all prompts with TAB(T) will be printed flush left, as long as the variable T is not used within your program. T will have a value of zero, and the program will tab zero spaces for each prompt.

When the program is done, save it in ASCII

```
10 PRINT TAB(T)"This program demonstrates the use"
20 PRINT TAB(T)"of TABBER.BAS.  Any program using"
30 PRINT TAB(T)"the special 'T' TAB will have that"
40 PRINT TAB(T)"prompt centered on the screen."
```

Fig. 4-1. Target Program to Demonstrate Tabber.

form, and run Tabber. It will search through each program line. When it finds TAB(T) it will measure the length of the prompt remaining, calculate how many spaces must be tabbed to center that message on a 40- or 80-column screen, and then replace the "T" with an appropriate number as shown in Fig. 4-2.

PROGRAMMING TIPS

A few programming techniques used in this program are described in this chapter. Menu input routines are one area ripe for improvement. Many programs will offer the operator a choice of actions, listed in a menu on the screen. Items from menus can be selected by having the user press the first letter of the menu item name, enter the whole choice, or enter a number that precedes the menu choice.

Having the user type in the whole name is rarely used, because a simple typing error could invalidate an otherwise correct entry. If a person wants a 40-column screen, and types 41 instead, it is a shame to make him or her redo the whole entry just for missing by one, or worse, having the program crash because it doesn't recognize the choice. Entering one character is popular, especially when a menu is accessed frequently. The user can easily memorize which letter triggers which menu choice, because of the mnemonic connection. The following is a typical letter-oriented menu:

```
(L)oad
(S)ave
(E)xit
(C)ontinue
```

A problem could occur if two menu choices started with the same letter, and the programmer could not think of a convenient synonym that used another initial letter. In addition, such menus force the nontypist user to hunt around the keyboard for letters that may be widely separated.

Numeric menus, on the other hand, have choices that are triggered by keys which are arranged in a row across the top of the keyboard. The limitation is that only ten menu choices can be listed, if you want single-key entry (0-9):

0.) Abort Operation
1.) Load
2.) Save
3.) Exit
4.) Continue

Even there, you open yourself to problems, because the simplest input methods could confuse a null entry (just pressing ENTER, for example) with zero. It is possible to check the CHR$ values of the entries, to differentiate between zero (CHR$(48)) and ENTER (CHR$(13)). You could also extend a numeric menu by using hexadecimal notation, following 9 with A,B,C,D, or E.

In practice, this is seldom needed. Tabber's menu has only two choices, that between 40- and 80-column formatting. It also uses a built-in error trap, something that is too often forgotten by beginning programmers. Some will write a routine like this:

```
10 PRINT"1.) Load program"
20 PRINT"2.) Save program"
30 INPUT"Enter Choice";CH
40 ON CH GOTO 100,200
```

Now, if a naive user enters L or S, or some other letter by mistake, a cryptic REDO FROM START message will be displayed. That is of no help at all. Entering a number larger than 2 will send the program to the line following 40, whatever *that* is. This could crash the whole program. You can avoid the REDO message by using CH$ instead of CH in the INPUT, since strings will accept letters as well as numbers. Converting to numerics, e.g., CH=VAL(CH$) will send you to our ON CH GOTO . . . line happily—except you still haven't handled the inappropriate input that might result. Also, it is necessary for the user to remember to hit the Enter key before the input is accepted. The user either has to be sophisticated enough to do this on his or her own, or else you have to waste one of the IBM PC's 25 screen lines to prompt the user to do so.

Since all you want is a single character, why not use INKEY$ to get it? Then, if the character is not valid, just send control back to the INKEY$ loop until a proper entry is made.

HOW TABBER WORKS

Tabber uses this INKEY$ loop approach. Line 220, for example, is an INKEY$ loop that repeats until a character is pressed. That character, A$, is converted to a number value, A, in line 230. If A<1 or A>2, the program loops back. Otherwise, it sets the value of S to either 40 or 80, as appropriate. (Variables used in Tabber are shown in Fig. 4-3.)

Next, the user enters the file names for the input and output files, and a single line is loaded from disk in line 330. The next line looks for an occurrence of "TAB(T)" in the target program line. Since the string "TAB(T)" is fairly unusual, no effort is made to check to see if it is contained in quotes or after a remark. Odds are that it will never appear in your program, except where you actually do want to center a prompt. This is mentioned because Tabber did "crash" when it was used to process itself. That is because of line 340, in which "TAB(T)" is contained as part of the program itself, and not before any prompt. In all other cases, TAB(T) will be followed by a prompt and a matched pair of quote marks. In this case, that was not so.

Whenever Tabber finds TAB(T), it looks for the position of the first quote, loads the value of the rest of the program line from that quote, and then cuts off the line AFTER the second quote (line 380). B$ will then contain only the material in the prompt.

The next step is to measure the length of the prompt, subtract that from S, which is the screen width (either 40- or 80-columns), and divide by 2. The resulting number, D, is the number of spaces that should be tabbed to center the prompt.

A new program line is then assembled in line 410, taking everything that appears BEFORE the TAB(, adding that to a string representation of the tab value (the leading space has been deleted in line 400), and finishing off with the rest of the program line, beginning with). Thus, the T has been deleted and replaced with a number. The program then loops back to line 340 to see if any more TAB(T)'s appear in the program line. This allows Tabber to process multiple TAB(T)'s appearing on a single line.

Once the work is finished, or if a line contains

```
10 PRINT TAB(23)"This program demonstrates the use"
20 PRINT TAB(23)"of TABBER.BAS.  Any program using"
30 PRINT TAB(23)"the special 'T' TAB will have that"
40 PRINT TAB(25)"prompt centered on the screen."
```

Fig. 4-2. Example of a program with TABs inserted.

A$	Program line being examined.
B$	Portion of program line.
C	Position of "TAB(T)" in program line.
C1	Position of quote in program line.
D	Half the difference between prompt length and display line length.
D$	Amount to tab, added to program line.
F$	File to be processed.
F2$	Output file.
S	Length of display line, either 64 or 80.

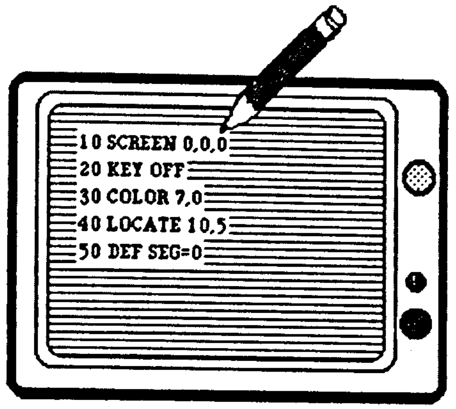
Fig. 4-3. Variables used by Tabber.

no TAB(T)'s in the first place, control drops down to lines 430-440, where A$ is printed to disk and screen. A check is made in line 420 to see if the end-of-file has been reached. If not, the program loops back to line 330 to load another program line from disk. Otherwise, the processing is finished.

Listing 4: The Tabber Program

```
10 ' ********************
20 ' *                  *
30 ' *    Tabber        *
40 ' *                  *
50 ' ********************

55 ' *** Initialize ***

60 KEY OFF
70 SCREEN 0,0,0
80 COLOR 7,0
90 ON KEY(10) GOSUB 550
100 KEY(10) ON
110 CLS:PRINT:PRINT
120 LOCATE 25,30
130 COLOR 16,7
140 PRINT" Hit F10 to abort. ";
150 COLOR 7,0
160 LOCATE 4,21
170 PRINT "IS PROGRAM FOR 40 OR 80 COLUMN SCREEN?"
180 PRINT
190 PRINT TAB(33)"1.) 40 COLUMN"
200 PRINT TAB(33)"2.) 80 COLUMN"
210 PRINT:PRINT TAB(33)"ENTER CHOICE :"
220 A$=INKEY$:IF A$="" GOTO 220
230 A=VAL(A$)
240 IF A<1 OR A>2 GOTO 220
250 IF A=1 THEN S=40 ELSE S=80

255 ' *** Enter Name of File to Process ***

260 CLS:PRINT:PRINT
270 PRINT TAB(20)"ENTER PROGRAM WITH TABS TO BE CENTERED:"
280 LINE INPUT F$
290 PRINT TAB(26)"ENTER NAME OF OUTPUT FILE :"
300 LINE INPUT F2$
310 OPEN "I",1,F$
320 OPEN "O",2,F2$

325 ' *** Load a Line ***

330 LINE INPUT #1,A$
340 C=INSTR(A$,"TAB(T)")
350 IF C=0 GOTO 430
360 C1=INSTR(C,A$,CHR$(34))+1
370 B$=MID$(A$,C1)
380 B$=LEFT$(B$,INSTR(B$,CHR$(34))-1)
390 D=INT((S-LEN(B$))/2)
400 D$=MID$(STR$(D),2)
410 A$=LEFT$(A$,C+3)+D$+MID$(A$,C+5)
420 GOTO 340

425 ' *** Print to Disk ***
430 PRINT #2,A$
440 PRINT A$
450 IF EOF(1) GOTO 470
460 GOTO 330
470 CLOSE
480 CLS:PRINT:PRINT
490 PRINT TAB(35)"FINISHED."

495 ' *** Again? ***

500 PRINT:PRINT
510 PRINT TAB(29)"Process another file?"
520 PRINT TAB(37)"(Y/N)"
530 A$=INKEY$:IF A$="" GOTO 530
540 IF A$="Y" OR A$="y" THEN RUN
550 CLOSE
560 CLS
570 END
```

Chapter 5



Screen Editor

The next three programs in the book, Screen Editor, DB Starter, and Proofer, make up a trilogy of sorts, called Automatic Programmer. The three in the Automatic Programmer series are related programs that might be thought of as integrated, but aren't. No data files are transferable from one to the other.

Output, however, from one of three can be processed or combined with output from the others quite easily.

These are an attempt to present some professional programming concepts, showing how error traps, help screens, instructional files, and so forth can enable programs to be self-documenting and usable even by the neophyte.

All three make use of a fourth program, Autoprogrammer Documentation, which serves as a help file and introduction to all three. It also is a menu of sorts that can be used to load and run one of the other programs.

THE PURPOSE OF SCREEN EDITOR

The first of the Automatic Programmer series is Screen Editor, which you will find to be one of the most useful programs in this book. I relied on it heavily to write instructional screens for many of the other programs here and even for itself. With a few minor changes, the program is compatible with the Microsoft Basic Compiler. A much faster running compiled version was used, cutting programming time down from a minute or two to a few seconds.

Have you ever wished that you could design your program menus, instruction screens, and other CRT displays with a word processor or some similar program—and then tell your IBM PC something like the following:

"Hey, I want my screen output to look like this. Please write a few lines of code for me that will reproduce this in my program."

Screen Editor will do exactly that for you. Use it as a screen-oriented text editor to lay out your display exactly as you want it to appear. Unlike an ordinary text editor, however, you can also use graphics! That is, you can take advantage of any of the ASCII special characters defined in the IBM include special border characters, musical notes, and foreign alphabets.

The program will then write a suitable subroutine, such as the one shown in Fig. 5-1, that can be MERGEd with an existing program to produce the desired display.

Ordinary, line-oriented program input and editing is somewhat tedious when neat, nicely formatted screen layout is desired. It's necessary to use a copy of the IBM PC screen map,and do a great deal of laborious notation on LOCATE positions. Even less complicated layouts require calculating TAB positions and other time-consuming tasks. Consider the work that would be involved in programming a display to provide the following menu:

```
****************************************
*                                      *
*              — Menu —                *
*                                      *
*         1.) Load disk file           *
*         2.) Save disk file           *
*         3.) Create file              *
*         4.) Access database          *
*         5.) Update database          *
*                                      *
*        —> Enter choice :             *
*                                      *
****************************************
```

With Screen Editor, simply use the arrow keys to move the cursor around on the full screen. Press character keys to place alphanumerics where desired. The layout can be quickly done by eye. Then, press the Enter key, specify what line numbers are desired for this subroutine, and collect the finished program module from your disk a few minutes later. There, stored in ASCII form (ready for merging) will be program lines that reproduce what you designed on the screen. Instead of spending 15 or 20 minutes of coding, RUNing the program to check the appearance of the output, making changes, and so forth, you have three to five minutes of typing with a word processor-like tool.

```
20 CLS
30 PRINT TAB(13)"****************************************"
40 PRINT TAB(13)"*
50 PRINT TAB(13)"*                                      *"
60 PRINT TAB(13)"*                                      *"
70 PRINT TAB(13)"*          SCREEN  EDITOR              *"
80 PRINT TAB(13)"*                                      *"
90 PRINT TAB(13)"*            FOR IBM PC                *"
100 PRINT TAB(13)"*                                     *"
110 PRINT TAB(13)"*                                     *"
120 PRINT TAB(13)"*                                     *"
130 PRINT TAB(13)"****************************************"
140 PRINT
150 PRINT TAB(20) "THIS IS A SCREEN PREPARED BY SCREEN EDITOR"
160 PRINT
170 PRINT
180 PRINT
190 PRINT
200 PRINT
210 PRINT
220 PRINT
230 PRINT
240 A$=INKEY$:IF A$="" GOTO 240
```

Fig. 5-1. An example of a program produced by Screen Editor.

HOW SCREEN EDITOR WORKS

This trick is accomplished by using the SCREEN function to check each position on the screen, noting what character (if any) has been placed there by the user, and then assigning each screen line to a separate element of a string array, L$(n). (See Fig. 5-2 for a list of the variables used in Screen Editor.) Then, each of the elements in L$(n) are used to assemble an appropriate program line, which PRINTs the entire line to the user's screen. If, say, line 1 consists of 10 spaces, 60 asterisks, and 10 more spaces, that entire line will be PRINTed in the resulting program. No calculations need to be made.

Screen Editor, in other words, reproduces your screen arrangement, spaces and all. It may not be the most memory efficient way of invoking a desired screen within your program, but for disk users with at least 64K of memory available, the waste will be negligible in comparison to the programming time saved.

Actually, a nifty technique is used to eliminate the leading and trailing spaces. As the program looks at each video line in turn, it sets a BFLAG when it encounters the first nonspace character, and an EFLAG when it encounters the LAST nonspace character on the line.

In assembling the finished program lines, it constructs a PRINT TAB statement that tabs to the position of the first nonspace. The following characters, spaces and all, are reproduced until the last nonspace, when a closing quote is added. Thus, a line like:

```
Hello!
```

Would not be turned into a program line like this:

```
10 PRINT"          Hello!
```

A$	Character input from keyboard, through INKEY$.
C	Cursor character.
CU	Counter.
EFLAG	End of character line flag.
F$	File name of output file.
IC	Increment to increase line number by.
L$	End of line character.
LN$(n)	Stores finished program lines.
LN$	Program line currently being built.
N	Loop counter.
N1-N9	Loop counters.
PR$	Program line being built.

Fig. 5-2. Variables used in Screen Editor.

Instead, the line would read:

```
10 PRINT TAB(10)"Hello!"
```

The program is divided into several sections. After going through some preliminary routines, it asks for a file name for the output file. An input routine used in several other programs in this book is accessed here. It starts at line 590.

The program puts the name you enter into variable F$. Then, a FOR-NEXT loop from 1 TO LEN(F$) looks at each character in the file name in turn. The ASC value of the character is calculated, and if it is greater than 96 and less than 123 (meaning it is a lowercase character), a conversion is made to uppercase. This is accomplished simply by subtracting 32 from the ASCII code for the character. That is, CHR$(97) ("a") becomes CHR$(65) ("A").

BASICA, of course, accepts a mixture of upper and lowercase in file names; however, I introduced this programming trick here because it will be used frequently later in the book, and converting the file name enables us to simplify some checking done later on.

For example, in line 640 the program looks to see if the first four characters of the file name are HELP or if the whole file name is H. This triggers a help routine. If the input had not been converted to uppercase, the program would have had to check for help, Help, hElp, helP and other combinations.

The routine looks to see if the file name is longer than 12 characters (eight characters plus a .BAS extension), and whether or not .BAS has been added. If it hasn't been, or if the file name is null, the name is rejected and the user asked to enter a new one. This routine will not catch ALL illegal file names, but it should keep the program from crashing because of the most common errors.

Next the program allows the user to input the screen design. An INKEY$ keyboard strobing loop subroutine looks for input (line 240). If F1 has been pressed, control drops down to the screen viewing/program assembly section. Otherwise, Screen Editor looks at the character input to see if it was Escape (CHR$(27)).

ESC is used as a *toggle* to turn graphics output on and off. A toggle is like an on/off pushbutton. If a feature is off when toggled, it is turned on. If it is on, then the toggle turns it off. Pressing ESC sets certain flags used by Screen Editor. If graphics mode was off, it is turned on by setting FLAG to 1 in line 1040.

The characters you press on the keyboard are printed to the screen. If FLAG=0, meaning graphics are off, the character, CU, will be the same as the key pressed. If graphics are switched on, CU will equal CU+128 (line 1070), and one of the characters from the second half of the IBM character set will be used. You should jot down which characters are produced by which keys when in graphics mode in order to use them in your own screen designs.

Screen Editor also checks to see if you press ENTER, CHR$(13). If you do, and the cursor is not already on line 24, then the cursor drops down to the beginning of the next line. That is done by changing the value of ROW and COL used with a LOCATE statement to print the cursor. In this case, COL is set to 1 (to move the cursor to the first column at the far left of the screen) and ROW is set to ROW+1, to move the cursor to the next row. This is not done when ROW=24, to avoid trying to move the cursor beyond the bottom of the screen. (Actually, what would happen would be that the screen would scroll, spoiling the design.)

Anytime the cursor reaches the middle of the screen, a BEEP is sounded. In addition, a display at the bottom of the screen tells you the current ROW, COL, graphics mode, graphics character, if any, and how to finish (with F1) or abort (with F10).

I've told you everything except how the cursor is moved on the screen. I used an interrupt routine, activated similarly to the "F10 to abort" routine used throughout this book. Instead of using one of the function keys to trigger the event, however, I used one of the cursor pad arrow keys.

These keys have definitions of their own. The IBM PC sees them as KEY(11) through KEY(14). So, when an arrow key is pressed, the program branches to a routine that changes the value of ROW or COL, either plus or minus, to move the cursor up, down, or from side to side. You should also know that BASIC 2.0 allows you to define any of the other keys on the keyboard as KEY(15) through KEY(20). You can even specify that ALT, CTRL, or SHIFT, or some combination of these be pressed. That technique, however, is not used here.

Please note that you must press the arrow keys once for each space you want to move. You cannot hold the key down. If you do, some graphics blocks may be printed to the screen, and you'll have to go back and erase them. The reason this takes place has to do with the way the IBM scans the keyboard and cannot be easily corrected through programming. REPEAT: Do not hold down the arrow keys.

Once the screen design is complete, and F1 is pressed, the program drops to line 1570. First, Screen Editor checks to see if a stray cursor character has been left on the screen and erases it (line 1580). Then the program starts two nested FOR-NEXT loops, the outer one covering each of the 24 screen lines and the inner one counting off each column across the screen. As the values for the loop counters change, SCREEN looks at each screen position and then paints it white. If the character is a nonspace, it is loaded into variable PR$, and used to construct the program line corresponding to that screen line. As mentioned, BFLAG and EFLAG mark the positions of the first and last nonspace characters. These variables are used to determine the initial TAB position of the line, and the place where the final quotation mark goes.

Actual program lines are written similarly to those in previous programs in this book. The program written is ended with an A$=INKEY$:IF A$=" " GOTO loop to keep your screen image displayed until you press a key.

Once Screen Editor has written a program to reproduce your screen design, you may MERGE it and edit it to suit yourself. Most of the screen designs in this book were written with Screen Editor, although in many cases they were edited to produce flashing characters and other special features.

Like all the programs in the Automatic Programmer series, Screen Editor has many error traps built in. Entering Help or H to the input prompts will call up the help file or display a tip. More complex error traps will be discussed in later chapters.

Listing 5: The Screen Editor Program

```
10 '      ***************************
20 '      *                         *
30 '      *       Screen Editor     *
40 '      *                         *
50 '      ***************************
60 '

65 ' *** Initialize ***

70 DEFINT A-Y
80 ROW=1:COL=1
90 WHITE=177
100 ON ERROR GOTO 2000
110 DIM LN$(400)
120 KEY OFF
```

```
130 SCREEN 0,0,0
140 COLOR 7,0
150 ON KEY(1) GOSUB 1560
160 ON KEY(10) GOSUB 2340
170 ON KEY(11) GOSUB 1500
180 ON KEY(12) GOSUB 1400
190 ON KEY(13) GOSUB 1350
200 ON KEY(14) GOSUB 1450
210 KEY(10) ON
220 WIDE=80
230 GOTO 260
240 A$=INKEY$:IF A$="" GOTO 240
250 RETURN
260 SP$=CHR$(32)
270 LA$=STRING$(64,"*")

275 ' *** Instructions? ***

280 CLS
290 GOSUB 400
300 GOTO 310
310 PRINT:PRINT
320 PRINT TAB(20)"-- Do you want general instructions ? --"
330 PRINT
340 PRINT TAB(16)"You may also type 'H' or 'HELP' to most input
prompts."
350 GOSUB 240
360 IF A$="N" OR A$="n" THEN CLS: GOTO 550
370 IF A$="H" OR A$="h" THEN RUN"AUTOPROG.BAS"
380 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG.BAS" ELSE 350
390 CLS
400 PRINT TAB(8)"********************************************
    ********************";
410 PRINT TAB(8)"*
                *";
420 PRINT TAB(8)"*
                *";                        Automatic Programmer
430 PRINT TAB(8)"*
                *";                         Screen Editor
440 PRINT TAB(8)"*
                *";                        By:     David D. Busch
450 PRINT TAB(8)"*
-------------------------------------------------------------------
460 PRINT                                                          *";
470 PRINT TAB(8)STRING$(64,"*")
480 RETURN
490 CLS
```

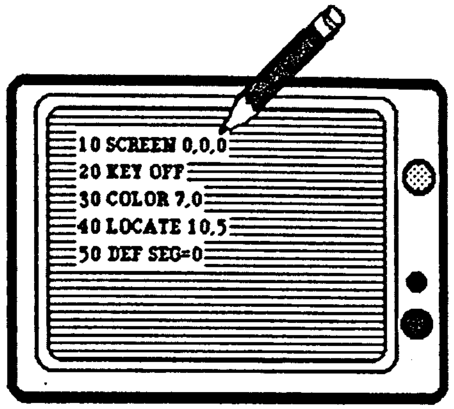
```
500 CLOSE
510 CU=1
520 :  FOR N8=1 TO 100
530 :    LN$(N8)=""
540 :  NEXT N8
550 LN=10:IC=10
560 PRINT:PRINT:PRINT
570 GOSUB 590
580 GOTO 710

585 ' *** Enter filename of screen ***

590 LINE INPUT"ENTER FILE NAME  :  ";F$
600 FOR N=1 TO LEN(F$)
610 T=ASC(MID$(F$,N,1))
620 IF T>96 AND T<123 THEN MID$(F$,N,1)=CHR$(T-32)
630 NEXT N
640 IF LEFT$(F$, 4)="HELP" OR F$="H" THEN GOSUB 2130
650 IF LEN(F$)>12 THEN PRINT"File name too long!":PRINT:GOTO 590
660 S9=INSTR(F$,".BAS")
670 IF LEN(LEFT$(F$,S9))>8 THEN PRINT"File name too
    long!":PRINT:GOTO 590
680 IF S9=0 THEN PRINT "MUST INCLUDE .BAS EXTENSION!":GOTO 590
690 IF F$="" GOTO 590
700 RETURN
710 IF F$="" THEN F$="TEST.BAS" :PRINT:PRINT
    "Using default filename TEST.BAS"

715 ' *** Instructions ***

720 CLS
730 PRINT:PRINT
740 PRINT TAB(11)"******************** Screen Editor
    ***********************"
750 PRINT TAB(11)"*
        * ";
760 PRINT TAB(11)"*  Use the cursor pad arrow keys to move
    around screen. * ";
770 PRINT TAB(11)"*  Press alphanumeric keys to type display.
    You may     * ";
780 PRINT TAB(11)"*  hit ESC, followed by a key to enter
    graphics mode.   * ";
790 PRINT TAB(11)"*  In graphics, press any key other than arrow
    keys to  * ";
800 PRINT TAB(11)"*  leave a trail of that graphics character.
    Use arrow * ";
```

```
810 PRINT TAB(11)"*  key to move without trail.  Exit graphics
    mode by     * ";
820 PRINT TAB(11)"*  hitting ESC once again to return to text,
    or to       * ";
830 PRINT TAB(11)"*  change to a different graphics character.
         * ";
840 PRINT TAB(11)"*
         * ";
850 PRINT TAB(11)"*  Computer will BEEP when cursor reaches
    center of the * ";
860 PRINT TAB(11)"*  screen.  Hit arrow keys once for each move;
    do NOT    * ";
870 PRINT TAB(11)"*  hold arrow key down, or graphics block will
    be left  * ";
880 PRINT TAB(11)"*  behind.  ONE key depression for each move
    only!       * ";
890 PRINT TAB(11)"*
         * ";
900 PRINT TAB(11)"******************************************
    ***************** ";
910 PRINT TAB(26)"-- HIT ANY KEY TO BEGIN -- "
920 GOSUB 240

925 ' *** Look for keyboard input ***

930 KEY(11) ON:KEY(12) ON:KEY(13) ON:KEY(14) ON
940 KEY(1) ON
950 CLS
960 GOSUB 240
970 A$=INKEY$:IF A$="" GOTO 970
980 IF A$<>CHR$(8) THEN GOTO 1030
990 COL=COL-1:IF COL<1 THEN COL=1
1000 LOCATE ROW,COL:PRINT CHR$(32);
1010 CU=0
1020 GOTO 1080
1030 IF A$=CHR$(27) AND FLAG2>0 THEN FLAG2=0:GOTO 1090
1040 IF A$=CHR$(27) THEN FLAG=1:GOTO 970
1050 IF A$=CHR$(13) AND ROW<24 THEN LOCATE ROW,COL:PRINT
     CHR$(32):ROW=ROW+1:COL=1
1060 CU=ASC(A$)
1070 IF FLAG=1 THEN CU=CU+128:FLAG=0:FLAG2=CU
1080 IF COL=WIDE/2 THEN BEEP
1090 LOCATE 25,1
1100 COLOR 0,7
1110 PRINT "Column: ";
1120 COLOR 7,0
1130 PRINT COL;
```

```
1140 LOCATE 25,15
1150 COLOR 0,7
1160 PRINT "Row : ";
1170 COLOR 7,0
1180 PRINT ROW;
1190 LOCATE 25,25
1200 COLOR 0,7
1210 PRINT" Graphics : ";
1220 COLOR 23,0
1230 IF FLAG2>1 THEN PRINT " ON "; ELSE COLOR 7,0:PRINT
     " OFF"+SPACE$(18);
1240 COLOR 7,0
1250 IF FLAG2>1 THEN COLOR 0,7:PRINT "Character :";:COLOR
     7,0:LOCATE 25,58:PRINT CHR$(FLAG2);:COLOR 7,0
1260 LOCATE 25,64:COLOR 16,7:PRINT"F1 TO PROCESS";:COLOR 7,0
1270 LOCATE ROW,COL
1280 IF VFLAG=1 THEN VFLAG=0:GOTO 1310
1290 IF FLAG2>0 THEN PRINT CHR$(FLAG2);:COL=COL+1:GOTO 1320
1300 IF CU>0 AND CU<>13 THEN PRINT CHR$(CU);:COL=COL+1
1310 IF CU=0 OR CU=13 THEN PRINT CHR$(43);
1320 CU=0
1330 GOTO 970

1335 ' *** Move Cursor ***

1350 LOCATE ROW,COL:PRINT CHR$(32);
1360 COL=COL+1
1370 IF COL>WIDE-1 THEN COL=WIDE-1
1380 VFLAG=1
1390 RETURN 1080
1400 LOCATE ROW,COL:PRINT CHR$(32);
1410 COL=COL-1
1420 IF COL<1 THEN COL=1
1430 VFLAG=1
1440 RETURN 1080
1450 LOCATE ROW,COL:PRINT CHR$(32);
1460 VFLAG=1
1470 ROW=ROW+1
1480 IF ROW>24 THEN ROW=24
1490 RETURN 1080
1500 LOCATE ROW,COL:PRINT CHR$(32);
1510 VFLAG=1
1520 ROW=ROW-1
1530 IF ROW<1 THEN ROW=1
1540 RETURN 1080
1550 GOTO 960
```

```
2310 PRINT TAB(29)"(Y/N)"
2320 A$=INKEY$:IF A$="" GOTO 2320
2330 IF A$="Y" OR A$="y" THEN RUN ELSE CLS
2340 CLOSE:END
```

Chapter 6



DataBase Starter

For the microcomputer user, the self-programming computer is still some time in the distant future. Or is it? There are three things that computers have a knack for, processing data, controlling functions, and constructing designs from smaller building blocks. The first two are simple enough. Ask a computer to add 367 to 598, and it will happily comply. Tell it to send a signal to port X whenever it receives input from port Y, and a computer will gladly control your carburetor, monitor your house, or keep your Boeing 767 on course. When a human is available to provide a list of criteria and parameters, a computer is entirely capable of combining components from an existing library to assemble or "design" a complex product.

A computer program is nothing more than a design to accomplish a desired task. Once a human being has determined how to get from point A to point B, it's entirely practical to have a computer choose from a library of subroutines to put together a program. The next program in the Automatic Programmer series is DB Starter, which illustrates the basic concept.

CREATING PROGRAM SKELETONS WITH DB STARTER

This program will ask the user for certain program parameters, such as whether or not a menu is needed, whether or not data will be stored in a string array, the size of the array, and other information, and then "write" a BASIC program skeleton that conforms to these parameters.

Figure 6-1 is a sample program that was written by DB Starter. The array in line 40 of the example was created and DIMensioned according to user input requirements, just as the menu was constructed, and subroutines allocated for later work by the human programmer. Two subroutines relating to disk I/O were actually entirely written by the program. The finished code was then saved to disk.

As written, the program will do the following things:

- ☐ Ask the user for beginning line number and desired line number increments.

```
30  DATA Name,Address,Phone,Zip
40  DIM DA$( 20, 30),DTA$( 4)
50  NC= 30
60  FOR G=1 TO  4:READ DTA$(G):NEXT G
70  CLS:PRINT:PRINT"    **********  MENU ***********":PRINT
80  PRINT" 1.)   Access Data"
90  PRINT" 2.)   Update Data"
100  PRINT" 3.)    Start Database"
110  PRINT" 4.)  LOAD FILE FROM DISK"
120  PRINT" 5.)  SAVE FILE TO DISK"
130  PRINT
140  INPUT"ENTER CHOICE : ";CH$
150  CH=VAL(CH$): IF CH<1 OR CH>  5 GOTO 140
160  ON CH GOSUB 500, 1000, 1500, 2000, 2500
170  GOTO  70
500   REM ****** INSERT Access Data SUBROUTINE HERE  ******
1000   REM ****** INSERT Update Data SUBROUTINE HERE  ******
1500   REM ****** INSERT Start Database SUBROUTINE HERE  ******
2000   REM ****** LOAD FILE FROM DISK ******
2010  INPUT "ENTER FILE NAME :";F$
2020   OPEN "I",1,F$
2030   INPUT #1,NF
2040  FOR N=1 TO NF
2050  FOR COL=1 TO NC
2060  INPUT #1,DA$(N,COL)
2070  NEXT COL,N
2080  CLOSE
2090  RETURN
2500   REM ****** SAVE FILE TO DISK ******
2510  INPUT "ENTER FILE NAME :";F$
2520   OPEN "O",1,F$
2530   PRINT #1,NF
2540  FOR N=1 TO NF
2550  FOR COL=1 TO NC
2560  PRINT #1,DA$(N,COL);","
2570  NEXT COL,N
2580  CLOSE
2590  RETURN
2600     REM ****** CLEAR SCREEN SUBROUTINE ******
2610  CLS:PRINT:PRINT:RETURN
2620    REM ****** INKEY$ INPUT SUBROUTINE *****
2630  A$=INKEY$:IF A$="" GOTO  2630
2640  A=VAL(A$)
2650  RETURN
```

Fig. 6-1. An example of a program produced by DB Starter.

□ Ask if a string array will be used to store data, and if so, allow the user to specify whether the array will be one- or two-dimensional. The elements that should be DIMensioned are also input.

□ A menu of reasonable size (i.e., which can fit on a single screen) may be specified. Each choice can be described. Program lines to print the menu to the screen will be created, along with an "enter choice" prompt.

□ Each of the menu choices will be assigned a subroutine line number—marked with a REMark—so the programmer can flesh them out later. An ON CH GOSUB . . . line will be created sending control to each of the menu subroutines.

□ Disk file I/O subroutines that will save or load data stored in a one- or two-dimensional array are automatically created.

□ The user can also specify several other subroutines, such as CLS:PRINT:PRINT and A$=INKEY$: IF A$=" " GOTO . . .

DB Starter will then create the basics of a simple data base management program that must be completed by the programmer. It doesn't complete the program, but does save a great deal of typing time. Arguably, there is a much simpler way of accomplishing nearly the same thing. Write out an all-purpose program containing the most-used modules and then SAVE that program on a convenient disk. When the time comes to create a new program, you can simply load the general module, delete lines not needed, renumber, and do other minor work to tailor it into a skeleton for the new project. Or you can use structured programming techniques with common variable names, routines, and so on to build a great many program modules that can be readily transferred from one program to another.

WHO NEEDS DB STARTER

Programs that write other programs make the most sense when developed for the unsophisticated user. That might include someone who is incapable of taking an all-purpose program and changing the code to fit a new purpose—a nonprogrammer, or a beginning programmer. Given a sufficiently sophisticated version of DB Starter, the user might be able to answer a series of prompts to inform the computer just what type of task had to be performed and then receive a finished program that will do the job.

DB Starter can only do a few things. While keeping the size of the program down to what will comfortably fit in this book, I've left the door open for ambitious programmers to expand its capabilities and apply the concepts to their own work.

HOW DB STARTER WORKS

Let's look at how the program works. The variables used are shown in Fig. 6-2. DB Starter consists of a series of modules, each designed to "create" a specific type of BASIC code. The mechanics are simple. The lines of the target program are assembled from the "library" of words and phrases built into DB Starter. As each line of the target program is completed, it is stored in a string array, LN$(n). The particular element of LN$(n) is determined by a counter, CU.

Each time a new target program line is initiated, control is sent to a subroutine at line 670. There, the line number of the target (LN) is incremented by IC (LN=LN+IC). IC is defined as 10 in line 490; however, you can change this to some other value or add an INPUT statement to permit the user to enter an increment at runtime. Next CU is increased by one so that the new program line will be stored in the next available element of LN$(n). Finally, the new line number (LN) is converted into a string and assigned as the first part of LN$(CU), along with a pair of spaces.

For example, if LN=100 and IC=10 when control is sent to line 670 of DB Starter, LN$(CU) will equal "110 " when it RETURNs. So, each element of LN$(n) will begin with a line number,

A$	Character input from keyboard through INKEY$.
CFLAG	Check to see end of DATA input.
CH$	User choice input.
COL$	Number of elements in second dimension of array.
CU	Counter.
D3$	Data string.
D4	Number of data items entered by user.
DI	Choice entered by user.
F$	File name for output file.
IC	Increment for line numbers.
IOFLAG	Whether or not user will need IO routines.
LN$(n)	Program lines being built.
MENU$(n)	Label for menu choices.
MI	Number of choices to be on menu.
N	Loop counter.
N1-N9	Loop counters.
NW	Loop counter.
P$	Substring of program line.
P1$	Substring of program line.
ROW$	Number of rows in user array.
Y$	Middle part of string.

Fig. 6-2. Variables used in DB Starter.

usually larger by IC from the previous element. The exception is when LN has been given a different value somewhere else in the program.

First, DB Starter asks the user whether or not some DATA lines should be written. You can enter the data elements consecutively, separated by commas. It is not necessary to enter line numbers or the word DATA at the beginning of each line. F1 is pressed when the DATA is finished.

A different INPUT routine is used here, beginning at line 840. We can't use INPUT, because that statement won't accept a comma in an entry. LINE INPUT is unsuitable in this case, because it will accept any key, including F1 as an entry, and we want to use F1 to signal when the DATA are finished. If LINE INPUT were used, the PC would pause and wait until the Enter key was pressed before activating the key-trapping routine.

So we use INKEY$, which will also accept commas as input, but which does not delay the triggering of the F1 key-interrupt. Any key pressed is added to the "answer," D3$, until the Enter key is pressed (just like LINE INPUT).

A check is made in line 1040 to make sure that any given DATA line does not end in a comma. A counter, D4, keeps track of how many DATA items have been entered. This is used later, when writing a READ DATA routine.

Next line 1140 asks the user whether or not a string array will be used to store data. If so, the number of dimensions are input into variable DI. If DI = 2, the user is asked to provide the desired size for each of the two dimensions (ROW and COL). If DI = 1, only ROW is used. The target program line is created by combining the line number (already stored in LN$(n), remember) with DIM, and the array dimensions, enclosed in parentheses. If a two-dimensional array has been specified, an additional line is developed that defines variable NC (number of columns) equal to COL. NC is used later in the target program to control disk input and output.

At this point, the program may create a line that looks like this:

```
150 DIM DA$(20,20)
```

If a menu is needed, DB Starter obligingly creates a line that labels one. Note that to make a PRINT statement, it is necessary to combine PRINT with quotation marks around the material to be printed. CHR$(34) (quotation marks) is stored in P1$, and this string variable used whenever quotation marks are needed in the target program.

The user is asked to input the number of choices required for the menu. If DI = 0 (that is, no string array was dimensioned), the program assumes that disk file I/O will not be required and does not offer the choice of taking advantage of the built-in disk I/O subroutines. Of course, disk files consisting of nothing but numeric values are possible. But the greater flexibility of storing both string and numeric data as strings (and then converting to numbers with VAL, as needed), makes it simpler for DB Starter to assume that disk files will be loaded into and out of a string array only.

If a string array has been specified, the user is asked if "Save file to disk" and "Load file from disk" will be included in the menu. If so, IOFLAG is set to 2. The user has told the program how many choices will be included on the menu. This value is transferred to CH, which is used as a parameter for a FOR-NEXT loop that allows input of the names of the menu choices.

If the built-in disk I/O routines are desired, two is subtracted from CH, so that the user does not have to bother to input these. That is, if five menu choices will be used, but two of them will be for disk I/O, the programmer has to enter only the other three. Then, the menu display lines are created for all but the disk routines.

Now things begin to get a little tricky. For each menu choice, the program has to create a subroutine location for the target program to branch to. Space has to be allocated for these. Instead of using LN, and incrementing it by IC, another variable, NU, is used. NU is incremented by IC*50 for each of the menu subroutines. For example, if IC = 10, then each of the subroutines will be spaced 500 lines apart from each other. The starting line numbers for each menu subroutine are stored in an array NU(n).

Next a string representation of each menu subroutine starting line number is needed (for an ON CH GOSUB 500, 1000, 1500, etc. statement). These are assembled with a comma tacked onto the end. Next, and INPUT "ENTER CHOICE :";CH$ line is created for the target program. An error trap is also built. When the target program is run, if VAL(CH$) is less than one or is greater than MI (the number of menu choices available), the input is refused.

All these subroutines in the ON CH GOSUB ... line will eventually RETURN, so control is sent back to the beginning of the menu. Its starting line number had been stored in IM(1) earlier and is used to build a control-branching instruction. To aid the programmer in finishing the skeletal program, a REM is inserted at each of the menu subroutine starting line numbers. Remember, it's not a good idea to send control to a REM line (these might be deleted later), so don't just begin writing the code at the next available line number following the remark.

The next portion builds a simple disk input module, which will ask the user for a file name, open that sequential file, input from the file the number of items in the file, and then begin a FOR-NEXT loop from 1 to the number of items in the file. Within the loop, INPUT #1 loads the data. If the relevant array is two-dimensional, a nested FOR-NEXT loop, from 1 to the number of columns

(NC—defined early in the program), is used. Actual construction of the disk input module is fairly clear-cut. Its mirror-image twin is the Create Disk Output routine, which performs its function in nearly the same manner.

ADDING YOUR OWN SUBROUTINES

Other modules that are frequently needed can be added to DB Starter's library as needed. I used a clear screen and INKEY$ routines as examples. You are free to add your own favorite subroutines as you desire. The final portion of the program saves the finished target program to disk under any desired legal name. A noncompressed (ASCII) file that can be loaded, finished, debugged, and used as desired is created.

DB Starter is simple enough to form the basis for a much more complex code-generating system. A big drawback is the need to anticipate just what capabilities will be needed in the finished program. If a subroutine isn't in the program generating system's library, or if the parameters are beyond its capabilities (i.e., a three-dimensional array is required), the necessary code will have to be built up from scratch.

It's still beyond the capability of microcomputers to use logic to create. Our silent servants must wait for instructions from us before doing anything at all, no matter how simple.

Listing 6: The DB Starter Program

```
10 '     ***************************
20 '     *
30 '     *                         *
40 '     *   DataBase Starter      *
50 '     *                         *
60 '     ***************************

65 ' *** Initialize ***

70 DEFINT A-Y
80 DIM LN$(400),NU(20)
90 KEY OFF
100 KEY 1,""
110 SCREEN 0,0,0
120 COLOR 7,0
130 ON KEY(1) GOSUB 3440
140 ON KEY(10) GOSUB 3460
150 KEY(10) ON
160 ON ERROR GOTO 2810
170 GOTO 200
180 A$=INKEY$:IF A$="" GOTO 180
190 RETURN
200 CU=1
210 P1$=CHR$(34)
220 P$=S2$+"PRINT"+S1$+P1$+S5$
230 CLS
240 GOSUB 330
```

```
250 GOTO 260

255 ' *** Instructions? ***

260 PRINT TAB(14)"-- Do you want general instructions ? --"
270 PRINT TAB(12)"You may also type 'H' or 'HELP' to most input
    prompts."
280 GOSUB 180
290 IF A$="N" OR A$="n" THEN CLS: GOTO 490
300 IF A$="H" OR A$="h" THEN RUN"AUTOPROG.BAS"
310 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG.BAS" ELSE 280
320 CLS
330 LOCATE 2,15
340 COLOR 0,7
350 PRINT" Automatic Programmer  --  DB Starter "
360 COLOR 7,0
370 PRINT
380 RETURN
390 CLS
400 CLOSE
410 CU=1:NU=1
420 :  FOR N=1 TO 20
430 :    NU(N)=0
440 :  NEXT N
450 NU$=""
460 :  FOR N8=1 TO 100
470 :    LN$(N8)=""
480 :  NEXT N8
490 LN=10:IC=10
500 PRINT:PRINT:PRINT
510 GOSUB 530
520 GOTO 650

525 ' *** Enter file name of program ***

530 LINE INPUT"ENTER FILE NAME  :  ";F$
540 FOR N=1 TO LEN(F$)
550 T=ASC(MID$(F$,N,1))
560 IF T>96 AND T<123 THEN MID$(F$,N,1)=CHR$(T-32)
570 NEXT N
580 IF LEFT$(F$, 4)="HELP" OR F$="H" THEN GOSUB 2920
590 IF LEN(F$)>12 THEN PRINT"File name too long!":PRINT
    :GOTO 530
600 S9=INSTR(F$,".BAS")
610 IF LEN(LEFT$(F$,S9))>8 THEN PRINT"File name too
    long!":PRINT:GOTO 530
```

```
630 IF F$="" GOTO 530
640 RETURN
650 IF F$="" THEN F$="TEST"
660 GOTO 710

665 ' *** Increment line number ***

670 LN=LN+IC
680 CU=CU+1
690 LN$(CU)=STR$(LN)+"  "
700 RETURN

705 ' *** Start writing program ***

710 CLS:PRINT:PRINT

715 ' *** Data Lines ***

720 PRINT"Would you like to build some data lines?"
730 PRINT
740 PRINT TAB(18)"Enter Y/N or ";CHR$(34);"H";CHR$(34);"(HELP)"
750 GOSUB 180
760 IF A$="H" OR A$="h" THEN GOSUB 3390: GOTO 710
770 IF A$="N" OR A$="n" GOTO 1130
780 IF A$="Y" OR A$="y" GOTO 840
790 GOTO 750
800 CU=CU+1
810 GOSUB 670
820 LN$(CU)=LN$(CU)+"DATA "
830 RETURN
840 KEY(1) ON
850 PRINT"Enter data elements to be written into program."
860 PRINT"Separate with commas.  Input no more than two lines"
870 PRINT"of DATA, then hit ENTER and input another pair of
    lines."
880 PRINT "It is not necessary to enter the word DATA.  Enter
    in"
890 PRINT"this form:  35,20,Address,Phone,Zip "
900 D3$=""
910 LOCATE 20,1:PRINT" Enter ";CHR$(34);"F1";CHR$(34);" to
    finish.":LOCATE 21,1:PRINT"Enter your DATA :"
920 LOCATE 22,5
930 PRINT SPACE$(60)
940 LOCATE 22,5
950 PRINT D3$;
960 A$=INKEY$:IF A$="" GOTO 960
970 IF A$=CHR$(8) AND D3$<>"" THEN
```

```
    D3$=LEFT$(D3$,LEN(D3$)-1):GOTO 920
980 IF A$=CHR$(13) THEN PRINT:GOTO 1020
990 D3$=D3$+A$
1000 PRINT A$;
1010 GOTO 960
1020 IF CFLAG=1 THEN GOTO 1130
1030 IF D3$="" THEN LOCATE 19,1:PRINT"You must enter data or
     ":FOR N=1 TO 1000:NEXT N:LOCATE 19,1:PRINT SPACE$(40);
     :GOTO 910
1040 IF RIGHT$(D3$, 1)="," THEN D3$=LEFT$(D3$, LEN(D3$)-1)
1050 FOR N7=1 TO LEN(D3$)
1060 Y$=MID$(D3$, N7, 1)
1070 IF Y$="," THEN D4=D4+1
1080 NEXT N7
1090 D4=D4+1
1100 GOSUB 800
1110 LN$(CU)=LN$(CU)+D3$
1120 IF CFLAG=0 GOTO 900

1125 ' *** Build arrays ***

1130 CLS:PRINT:PRINT
1140 PRINT"Will this program store disk I/O data in a string
     array?"
1150 GOSUB 180
1160 IF A$="H" OR A$="h" THEN GOSUB 3030: GOTO 1130
1170 IF A$="N" OR A$="n" THEN 1450
1180 IF A$="Y" OR A$="y" THEN 1190 ELSE 1150
1190 GOSUB 670
1200 PRINT"Will the array have one or two dimensions?"
1210 GOSUB 180
1220 IF A$="H" OR A$="h" THEN GOSUB 3030: GOTO 1200
1230 DI=VAL(A$)
1240 IF DI<1 OR DI>2 THEN 1210
1250 IF DI=1 GOTO 1350
1260 INPUT"How many elements in the first dimension (ROW)";ROW$
1270 IF LEFT$(ROW$, 1)="h" OR LEFT$(ROW$, 1)="H" THEN
     GOSUB 3140: GOTO 1260
1280 INPUT"Enter elements in second dimension (COL) :";COL$
1290 IF LEFT$(COL$, 1)="h" OR LEFT$(COL$, 1)="H" THEN
     GOSUB 3140: GOTO 1280
1300 ROW=VAL(ROW$)
1310 COL=VAL(COL$)
1320 IF ROW<1 THEN ROW=1
1330 IF COL<1 THEN COL=1
1340 GOTO 1390
1350 INPUT"How large should the array be";ROW$
```

```
1360 IF LEFT$(ROW$, 1)="H" OR LEFT$(ROW$, 1)="h" THEN
     GOSUB 3140: GOTO 1350
1370 ROW=VAL(ROW$)
1380 IF ROW<1 THEN ROW=1
1390 LN$(CU)=LN$(CU)+"DIM DA$(" +STR$(ROW)
1400 IF DI=1 THEN LN$(CU)=LN$(CU)+")": GOTO 1450
1410 LN$(CU)=LN$(CU)+"," +STR$(COL)+")"
1420 GOSUB 670
1430 LN$(CU)=LN$(CU)+"NC=" +STR$(COL)
1440 IF D4>0 THEN LN$(CU-1)=LN$(CU-1)+",DTA$(" +STR$(D4)+")"
     : GOTO 1460
1450 IF D4>0 THEN GOSUB 670: LN$(CU)=LN$(CU)+"DIM DTA$("
     +STR$(D4)+")"
1460 IF D4>0 THEN CU=CU+1: GOSUB 670: LN$(CU)=LN$(CU)+"FOR G=1
     TO " +STR$(D4)+":READ DTA$(G):NEXT G

1465 ' *** Build Menus ***

1470 PRINT"Will this program need a menu?"
1480 GOSUB 180
1490 IF A$="H" OR A$="h" GOTO 2980
1500 IF A$="N" OR A$="n" THEN 2030
1510 IF A$="y" OR A$="y" THEN PRINT A$: GOTO 1520 ELSE 1480
1520 GOSUB 670
1530 LN$(CU)=LN$(CU)+"CLS:PRINT:" +P$+"   **********  MENU
     **********" +P1$+":PRINT"
1540 IM(1)=LN
1550 CLS:PRINT:PRINT
1560 INPUT"How many choices on the menu";CH$
1570 IF LEFT$(CH$, 1)="H" OR LEFT$(CH$, 1)="h" THEN GOSUB 3190
     : GOTO 1550
1580 MI=VAL(CH$)
1590 IF MI<2 GOTO 1550
1600 IF DI=0 THEN 1680
1610 IF MI=2 THEN CH=MI: GOTO 1710
1620 PRINT"Will the choices include 'Save file to disk' and
     'Load file from disk' ? ";
1630 GOSUB 180
1640 IF A$="H" OR A$="h" THEN GOSUB 3300: GOTO 1620
1650 IF A$="Y" OR A$="y" THEN IOFLAG=2: PRINT A$: GOTO 1680
1660 IF A$="n" OR A$="N" THEN PRINT A$: GOTO 1680
1670 GOTO 1630
1680 CH=MI
1690 IF CH=IOFLAG THEN N=1: GOTO 1800
1700 CH=CH-IOFLAG
1710 :   FOR N=1 TO CH
1720 :     PRINT"Enter label for menu choice #";N
```

```
1730 :     INPUT MENU$(N)
1740 :     IF MENU$(N)="HELP" OR MENU$(N)="H" OR MENU$(N)="h"
           THEN GOSUB 3270: GOTO 1720
1750 :   NEXT N
1760 :   FOR NW=1 TO CH
1770 :     GOSUB 670
1780 :     LN$(CU)=LN$(CU)+P$+STR$(NW)+".)    " +MENU$(NW)+P1$
1790 :   NEXT NW
1800 IF IOFLAG=2 THEN GOSUB 670: LN$(CU)=LN$(CU)+P$+STR$(N)+".)
     " +"LOAD FILE FROM DISK" +P1$: GOSUB 670:
     LN$(CU)=LN$(CU)+P$+STR$(N+1)+".)   " +"SAVE FILE TO DISK"
     +P1$
1810 GOSUB 670
1820 LN$(CU)=LN$(CU)+"PRINT"
1830 :   FOR NW=1 TO MI
1840 :     NU=NU+IC*50
1850 :     NU(NW)=NU
1860 :     NU$=NU$+STR$(NU)+","
1870 :   NEXT NW
1880 NU$=LEFT$(NU$,(LEN(NU$)-1))
1890 GOSUB 670
1900 LN$(CU)=LN$(CU)+"INPUT" +P1$+"ENTER CHOICE : " +P1$+";CH$"
1910 GOSUB 670
1920 LN$(CU)=LN$(CU)+"CH=VAL(CH$): IF CH<1 OR CH> " +STR$(MI)+"
     GOTO" +STR$(VAL(LN$(CU-1)))
1930 GOSUB 670
1940 LN$(CU)=LN$(CU)+"ON CH GOSUB" +NU$
1950 GOSUB 670
1960 LN$(CU)=LN$(CU)+"GOTO " +STR$(IM(1))
1970 :   FOR N=1 TO MI-IOFLAG
1980 :     GOSUB 670
1990 :     LN=NU(N)
2000 :     LN$(CU)=STR$(NU(N))+"   REM ****** INSERT "
           +MENU$(N)+" SUBROUTINE HERE " +" ******"
2010 :   NEXT N
2020 IF IOFLAG<>2 THEN 2510
2030 GOSUB 670
2040 PRINT
2050 IF MI=0 THEN LN$(CU)=LN$(CU)+"  REM ****** LOAD FILE FROM
     DISK": GOTO 2080
2060 LN=NU(N)
2070 LN$(CU)=STR$(NU(N))+"   REM ****** LOAD FILE FROM DISK
     ******"
2080 GOSUB 670
2090 LN$(CU)=LN$(CU)+"INPUT " +P1$+"ENTER FILE NAME :"
     +P1$+";F$"
2100 GOSUB 670
```

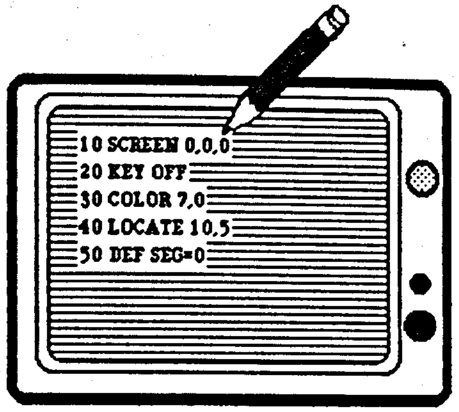
```
2110 LN$(CU)=LN$(CU)+" OPEN " +P1$+"I" +P1$+",1,F$"
2120 GOSUB 670
2130 LN$(CU)=LN$(CU)+" INPUT #1,NF"
2140 GOSUB 670
2150 LN$(CU)=LN$(CU)+"FOR N=1 TO NF"
2160 GOSUB 670
2170 IF DI=2 THEN LN$(CU)=LN$(CU)+"FOR COL=1 TO NC": GOSUB 670
2180 LN$(CU)=LN$(CU)+"INPUT #1,DA$(N"
2190 IF DI=2 THEN LN$(CU)=LN$(CU)+",COL)" ELSE
     LN$(CU)=LN$(CU)+")"
2200 GOSUB 670
2210 LN$(CU)=LN$(CU)+"NEXT"
2220 IF DI=2 THEN LN$(CU)=LN$(CU)+" COL,N"
2230 GOSUB 670
2240 LN$(CU)=LN$(CU)+"CLOSE"
2250 GOSUB 670
2260 LN$(CU)=LN$(CU)+"RETURN"
2270 GOSUB 670
2280 IF MI=0 THEN LN$(CU)=LN$(CU)+"   REM ****** SAVE FILE TO
     DISK ******": GOTO 2310
2290 LN=NU(N+1)
2300 LN$(CU)=STR$(NU(N+1))+"   REM ****** SAVE FILE TO DISK
     ******"
2310 GOSUB 670
2320 LN$(CU)=LN$(CU)+"INPUT " +P1$+"ENTER FILE NAME :"
     +P1$+";F$"
2330 GOSUB 670
2340 LN$(CU)=LN$(CU)+" OPEN " +P1$+"O" +P1$+",1,F$"
2350 GOSUB 670
2360 LN$(CU)=LN$(CU)+" PRINT #1,NF"
2370 GOSUB 670
2380 LN$(CU)=LN$(CU)+"FOR N=1 TO NF"
2390 GOSUB 670
2400 IF DI=2 THEN LN$(CU)=LN$(CU)+"FOR COL=1 TO NC": GOSUB 670
2410 LN$(CU)=LN$(CU)+"PRINT #1,DA$(N"
2420 IF DI=2 THEN LN$(CU)=LN$(CU)+",COL)" ELSE
     LN$(CU)=LN$(CU)+")"
2430 LN$(CU)=LN$(CU)+";" +P1$+"," +P1$
2440 GOSUB 670
2450 LN$(CU)=LN$(CU)+"NEXT"
2460 IF DI=2 THEN LN$(CU)=LN$(CU)+" COL,N"
2470 GOSUB 670
2480 LN$(CU)=LN$(CU)+"CLOSE"
2490 GOSUB 670
2500 LN$(CU)=LN$(CU)+"RETURN"

2505 ' *** Subroutines ? ***
```

```
2510 PRINT"Do you want a 'CLEAR SCREEN' subroutine?  ";
2520 GOSUB 180
2530 IF A$="H" OR A$="h" THEN GOSUB 3360: GOTO 2510
2540 IF A$="Y" OR A$="y" THEN PRINT A$: GOTO 2570
2550 IF A$="N" OR A$="n" THEN PRINT A$: GOTO 2610
2560 GOTO 2520
2570 GOSUB 670
2580 LN$(CU)=LN$(CU)+"   REM ****** CLEAR SCREEN SUBROUTINE
     ******"
2590 GOSUB 670
2600 LN$(CU)=LN$(CU)+"CLS:PRINT:PRINT:RETURN"
2610 PRINT"Do you want an 'INKEY$-INPUT' subroutine?  ";
2620 GOSUB 180
2630 IF A$="H" OR A$="h" THEN GOSUB 3360: GOTO 2610
2640 IF A$="N" OR A$="n" THEN PRINT A$: GOTO 2750
2650 IF A$="Y" OR A$="y" THEN PRINT A$: GOTO 2670
2660 GOTO 2620
2670 GOSUB 670
2680 LN$(CU)=LN$(CU)+"  REM ****** INKEY$ INPUT SUBROUTINE
     *****"
2690 GOSUB 670
2700 LN$(CU)=LN$(CU)+"A$=INKEY$:IF A$=" +P1$+P1$+" GOTO "
     +STR$(LN)
2710 GOSUB 670
2720 LN$(CU)=LN$(CU)+"A=VAL(A$)"
2730 GOSUB 670
2740 LN$(CU)=LN$(CU)+"RETURN"

2745 ' *** Write program to disk ***

2750 OPEN"O",1, F$
2760 :  FOR N1=1 TO CU
2770 :    PRINT#1, LN$(N1)
2780 :  NEXT N1
2790 CLOSE
2800 RUN

2805 ' *** Error Trap ***

2810 PRINT:PRINT
2820 PRINT TAB(20)"*****  UNKNOWN ERROR  *****"
2830 PRINT TAB(25)"IN LINE ";ERL
2840 FOR N9=1 TO 500
2850 NEXT N9
2860 CLS:PRINT:PRINT
2870 RETURN
2880 PRINT
```

```
2890 PRINT TAB(15)"Hit any key to resume program"
2900 GOSUB 180
2910 RETURN

2915 ' *** Help Routines ***

2920 GOSUB 2860
2930 PRINT"You should enter the filename you want  -- it must"
2940 PRINT"be a legal Disk basic name, or your input will be"
2950 PRINT"rejected."
2960 PRINT
2970 RETURN 530
2980 GOSUB 2860
2990 PRINT"Menus may be designed using a special"
3000 PRINT"module that asks for number of choices, labels,etc."
3010 GOTO 1470
3020 GOTO 2880
3030 GOSUB 2860
3040 PRINT"Many forms of data are conveniently stored in a
     string"
3050 PRINT"array which looks like this: DA$(row,col).  A
     checkbook"
3060 PRINT"represents data that can be stored in a
     two-dimensional"
3070 PRINT"array.  Each check number represents a row, while
     payee"
3080 PRINT"amount, balance, etc. represent columns.  These
     arrays"
3090 PRINT"can be conveniently stored and loaded to and from
     disk."
3100 PRINT"Use a one-dimensional  array for information which
     has only"
3110 PRINT"one 'field' per record.  If rows and columns are
     involved"
3120 PRINT"use a two dimensional array."
3130 GOTO 2880
3140 GOSUB 2860
3150 PRINT"Enter how large each dimension of the array should
     be"
3160 PRINT"For example, might want an array: DA$(30,30)."
3170 PRINT"Do not make much larger than you need to save
     memory."
3180 GOTO 2880
3190 GOSUB 2860
3200 PRINT"Most programs with multiple functions need a menu so
     the"
```

```
3210 PRINT"user may choose.  Automatic Programmer will design a
     menu"
3220 PRINT"for you and write apprcpriate  input and error
     trapping"
3230 PRINT"routines.  Or, you may design your own menu.  You
     must"
3240 PRINT"then write your own input routine, or use the INKEY$"
3250 PRINT"subroutine provided."
3260 GOTO 2880
3270 GOSUB 2860
3280 PRINT"Enter the label or prompt for this menu choice :"
3290 GOTO 2880
3300 GOSUB 2860
3310 PRINT"If you have specified a string array, and need disk
     I/O"
3320 PRINT"You should enter Yes.  Program will write these
     routines"
3330 PRINT"for you, and reduce number of menu choices you have"
3340 PRINT"to input by two.  Menu labels will be created for
     you."
3350 GOTO 2880
3360 GOSUB 2860
3370 PRINT"Enter Yes if you want this subroutine in your
     program"
3380 GOTO 2880
3390 GOSUB 2860
3400 PRINT"You may build data lines automatically, along with"
3410 PRINT"a routine to read them into a string array.  Just"
3420 PRINT"enter the data information when asked"
3430 GOTO 2880
3440 CFLAG=1
3450 KEY(1) OFF: RETURN 1020
3460 CLOSE:END
```

Chapter 7



Program Proofer

In the two previous Automatic Programmer examples, I've shown you how to let your computer write its own screens and assemble program skeletons. Now, here's Program Proofer, which allows an IBM PC to partially debug its own programs by checking the spelling of keywords and some syntax errors.

Some program errors caused by misspelled words lurk deep within seldom called code. Ordinarily, obvious bugs will surface during program development, because the interpreter will note a syntax error when the line is run. The experienced programmer will try to test a program with all possible conditions and parameters in order to give each section of code a workout. Program subroutines should be tested individually and when combined with the main program.

In the real world, however, such thorough testing is not always done. Errors will not be detected for some time, because the specific conditions that invoke those program lines are rare. In the worst possible situations, these mistakes are hidden in error traps designed to help the unsophisticated user, or they may cause the loss of valuable data. Program Proofer will check every line of a program and detect all bad keywords. It will catch only typos, however. If you used LPRINT when you meant PRINT, the bug will slip by unchecked.

HOW PROGRAM PROOFER WORKS

Program Proofer was inspired by the plethora of spelling checker programs that have become available in the past few years. These useful software tools take any text document and compare each word against an internal dictionary. Any word in your text that does not appear in the dictionary is flagged as a possible spelling error.

This program works on exactly the same principle, but with a much smaller dictionary of 172 keywords. These are the reserved words named by IBM in the BASIC user's manual. Some are commands or functions that are not implemented, but all were included to make this program compatible with later releases of DOS and BASIC.

Program Proofer examines every word in a target program. It ignores words inside quotes—prompts, for example—numbers, and arithmetic operators. The only letter combinations that are left are keywords, variables (Fig. 7-1), and misspelled words. Although it would be possible to tell which of the remaining words are variables—leaving only the incorrect keywords—I decided not to implement this feature. As written, Program Proofer has the added capability of providing a variable cross-reference listing that includes line numbers.

Not throwing out variables also means that the operator has the opportunity to look for variables that may have been spelled wrong, as well. This is important to IBM PC users. Under some versions of Microsoft Basic for other computers, PREVIOUS and PREVIUS would appear as the same variables, although PREVIOUS and PEVIOUS would not. With those Microsoft BASICs, only the first two letters of the variable name are significant. So, finding such misspellings is important.

With the IBM PC, however, longer variable names are allowed, and so finding errors is even more important. PREVIOUS and PREVIUS would, in fact, be different variables and cause an error if the difference was unintended.

This program will handle most ASCII format BASIC programs. Multiple statements per line are okay. Keywords should have spaces separating them, and there should be a space after the line number and before the first word in the line. These spaces are required by IBM BASIC, anyway, but Proofer needs them in order to find other errors.

When asked for the target program name, enter the file specification of the previously saved ASCII format program. It will be stored in the variable F$.

A$	Line of text being proofed.
BAD$(n)	Array storing bad words and variables.
D$	Temporarily stores good keyword names.
D2	ASCII value of first character in keyword.
F$	File name of program being proofed.
L	Length of the program segment being proofed.
LP	Number of left parentheses found.
M$	Middle string of SEG$.
N	Loop counter.
N1-N9	Loop counters.
NI	Counter.
NU	Counter.
P	Position of space in program line being checked.
PAR$(n)	Lines with odd number of parentheses.
PFLAG	Send output to printer.
SEG$	Program segment being proofed.
TEST$	Program segment being tested.
WRD$(n,nl)	Array storing good keywords.
Z3	Number of line printed.
ZU	Number of lines printed.

Fig. 7-1. Variables used in Program Proofer.

(For a complete list of the variables used in Proofer, see Fig. 7-1.) Each line in the target program will be examined separately, and all words not included within quotation marks compared with the internal dictionary. If a match is not found, the questionable word (which may also be a variable) is stored away for later reference. The number of parentheses are counted, and any missing ones noted. Program Proofer will also locate absent quotation marks, and list all the variables used in the program. In all cases, line numbers are provided to make tracking down the errant bugs easier.

Here, briefly, is how Program Proofer works. The 172 keywords are stored in a string array, WRD$(26,30). Each of 26 rows in the array correspond to one of the 26 letters of the alphabet. The 30 columns allow for up to 30 keywords beginning with that letter. For example, ABS is stored in WRD$(1,1), while AND is placed in WRD$(1,2). This is accomplished in a FOR-NEXT loop beginning at line 770. The keyword is read from a data line, and the first letter examined to determine its ASCII value. Then 64 is subtracted to arrive at the alphabetic position and the corresponding ROW of WRD$(row,col). CDBL, which begins with C (ASCII 67), is directed to Row 3 (67 minus 64). The column is determined by a counter, A, which is incremented every time a new keyword is READ, and reset to one each time a new ROW is opened (A2 < > PREVIOUS.)

As IBM BASIC expands with new features, statements, and functions, Program Proofer may be updated to include these new keywords and commands. Add the word to the proper position in the DATA lines and change the 172 to the new numbers of keywords. If a given letter of the alphabet now has more than 30 keywords, it will be necessary to reDIMension WRD$(row,col) as well.

The target program (F$) is OPENed, and a line at a time LINE INPUT into variable A$. The first space in the program line is assumed to follow the line number, and the rest of the line is stored in SEG$. A FOR-NEXT loop from one to L+1 (length of SEG$) examines each character in the program line in turn.

When certain delimiters are reached, the program assumes that the end of a word or variable has been located. These delimiters include a space, quotation mark, comma, semicolon, parentheses, colon, and arithmetic signs such as plus, minus, equals, or less than. At this point, control drops to a subroutine, where that portion of the line, TEST$, is subjected to a series of tests.

If TEST$ = " " (null), or if the value of the first character is greater than zero (signifying a number), then the program jumps back and begins looking at the next section of the program line. This program won't accept a keyword or variable beginning with a number.

When "REM" or its abbreviation " ' " is encountered, the program knows that the rest of the program line should be ignored.

Once TEST$ gets past these checks, it enters a FOR-NEXT loop from one to 30, which compares TEST$ with all the elements of WRD$(row,col) beginning with the same letter of the alphabet as TEST$. If a match is found, FLAG is set to one, and control drops to 1210. If no match is found before the end of the list of keywords beginning with the appropriate letter is reached, FLAG is set to zero, and control drops to 1210, where a counter, NU, is incremented, and the suspect word stored in string array BAD$(n), along with the line number in which it appears. The word itself is positioned first, followed by the line number, so that the array may later be sorted into alphabetical order.

Then, whether or not there was a match, TEST$ is nulled, and the rest of the line looked at for additional statements, variables, and keywords.

Any time a quotation mark is encountered, SFLAG is set to one, and any additional characters in a line are ignored until the second ("close quote") is located. Then the following words are considered and checked normally. Though no specific check for missing quotation marks is built in, they will stand out like a sore thumb, because in the final listing, words inside prompts will be listed as bad words.

A check is included for absent parentheses, however. Each right parenthesis encountered in a program line increments variable RP, while left parentheses increase the value of LP by one. After the whole program line has been checked, Program Proofer compares LP and RP. If they don't match, the line in which the error appears is stored in a string array PAR$(n), along with a note as to whether it is a left or right parenthesis that is missing. Note: if one statement is missing a left parenthesis, while another statement later in that line is missing a right parenthesis, the LP and RP will match, and the error will not be caught. This should occur very rarely, however.

When the end of file (EOF) marker is encountered, the user is asked if results should be directed to a printer as well as to the screen. The suspect words are then printed out in groups of 16 word/line (each word occupies one line).

A counter, ZU, keeps track of how many words are printed or listed. A word/line combination is displayed only if it does not equal the previous word/line. So, if a variable or bad word appears several times in a single line, it is pointed out just once. When ZU can be evenly divided by 16, the program branches to a "paging" subroutine at line 1570.

Once the variables and bad words are listed, the program displays all the lines which contain missing parentheses.

POSSIBLE ENHANCEMENTS

A number of enhancements are possible. The program could be extended to check each variable against the keyword list, using INSTR, to see if you have inadvertently included a nonallowable keyword within a variable name.

Checking the spelling of a computer program is much easier than proofreading a document, because the number of legal words is severely limited. Once a computer is told what words are allowable in a program, it is a simple matter to leave some of the tedious debugging to the machine, which will benefit most from clear instructions.

Listing 7: The Program Proofer Program

```
10 '     ***************************
20 '     *
30 '     *                         *
40 '     *    Program Proofer      *
50 '     *                         *
60 '     ***************************

65 ' *** Initialize ***

70 DEFINT A-Y
80 KEY OFF
90 ON KEY(10) GOSUB 1940
100 KEY(10) ON
110 SCREEN 0,0,0
120 COLOR 7,0
130 ON ERROR GOTO 1700
140 DIM WRD$(26,30),PAR$(30),BAD$(200)
```

```
150 GOTO 180
160 A$=INKEY$:IF A$="" GOTO 160
170 RETURN
180 CLS
190 GOSUB 300
200 GOTO 210

205 ' *** Instructions ***

210 PRINT:PRINT
220 PRINT TAB(20)"-- Do you want instructions ? --"
230 PRINT
240 PRINT TAB(8)"   You may also type 'H' or 'HELP' to most
    input prompts."
250 GOSUB 160
260 IF A$="N" OR A$="n" THEN CLS: GOTO 360
270 IF A$="H" OR A$="h" THEN RUN"AUTOPROG.BAS"
280 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG.BAS" ELSE 250
290 CLS
300 PRINT TAB(T)"Automatic Programmer"
310 PRINT TAB(T)"PROGRAM PROOFER"
320 PRINT TAB(T)"By:    David D. Busch"
330 RETURN
340 CLS
350 CLOSE
360 PRINT:PRINT

365 ' *** Input filename to be proofed ***

370 LINE INPUT"ENTER FILE NAME  :  ";F$
380 FOR N=1 TO LEN(F$)
390 T=ASC(MID$(F$,N,1))
400 IF T>96 AND T<123 THEN MID$(F$,N,1)=CHR$(T-32)
410 NEXT N
420 IF LEFT$(F$, 4)="HELP" OR F$="H" THEN GOSUB 1870
430 IF LEN(F$)>12 THEN PRINT"File name too long!":PRINT:GOTO 370
440 S9=INSTR(F$,".BAS")
450 IF LEN(LEFT$(F$,S9))>8 THEN PRINT"File name too
    long!":PRINT:GOTO 370
460 IF S9=0 THEN PRINT "MUST INCLUDE .BAS EXTENSION!":GOTO 370
470 IF F$="" GOTO 370
480 RESTORE
490 DATA ABS,AND,ASC,ATN,AUTO,BEEP,BLOAD,BSAVE,CALL,CDBL
500 DATA CHAIN,CHDIR,CHR$,CINT,CIRCLE,CLEAR,CLOSE,CLS,COLOR
510 DATA COM,COMMON,CONT,COS,CSNG,CSRLIN,CVD,CVI,CVS
520 DATA "DATA",DATE$,DEF,DEFDBL,DEFINT,DEFSNG,DEFSTR
530 DATA DELETE,DIM,DRAW,EDIT,ELSE,END,ENVIRON,ENVIRONS,EOF
```

stand out like a sore thumb, because in the final listing, words inside prompts will be listed as bad words.

A check is included for absent parentheses, however. Each right parenthesis encountered in a program line increments variable RP, while left parentheses increase the value of LP by one. After the whole program line has been checked, Program Proofer compares LP and RP. If they don't match, the line in which the error appears is stored in a string array PAR$(n), along with a note as to whether it is a left or right parenthesis that is missing. Note: if one statement is missing a left parenthesis, while another statement later in that line is missing a right parenthesis, the LP and RP will match, and the error will not be caught. This should occur very rarely, however.

When the end of file (EOF) marker is encountered, the user is asked if results should be directed to a printer as well as to the screen. The suspect words are then printed out in groups of 16 word/line (each word occupies one line).

A counter, ZU, keeps track of how many words are printed or listed. A word/line combination is displayed only if it does not equal the previous word/line. So, if a variable or bad word appears several times in a single line, it is pointed out just once. When ZU can be evenly divided by 16, the program branches to a "paging" subroutine at line 1570.

Once the variables and bad words are listed, the program displays all the lines which contain missing parentheses.

POSSIBLE ENHANCEMENTS

A number of enhancements are possible. The program could be extended to check each variable against the keyword list, using INSTR, to see if you have inadvertently included a nonallowable keyword within a variable name.

Checking the spelling of a computer program is much easier than proofreading a document, because the number of legal words is severely limited. Once a computer is told what words are allowable in a program, it is a simple matter to leave some of the tedious debugging to the machine, which will benefit most from clear instructions.

Listing 7: The Program Proofer Program

```
10 '        ****************************
20 '        *                          *
30 '        *     Program Proofer      *
40 '        *                          *
50 '        ****************************
60 '

65 ' *** Initialize ***

70 DEFINT A-Y
80 KEY OFF
90 ON KEY(10) GOSUB 1940
100 KEY(10) ON
110 SCREEN 0,0,0
120 COLOR 7,0
130 ON ERROR GOTO 1700
140 DIM WRD$(26,30),PAR$(30),BAD$(200)
```

```
150 GOTO 180
160 A$=INKEY$:IF A$="" GOTO 160
170 RETURN
180 CLS
190 GOSUB 300
200 GOTO 210

205 ' *** Instructions ***

210 PRINT:PRINT
220 PRINT TAB(20)"-- Do you want instructions ? --"
230 PRINT
240 PRINT TAB(8)"   You may also type 'H' or 'HELP' to most
    input prompts."
250 GOSUB 160
260 IF A$="N" OR A$="n" THEN CLS: GOTO 360
270 IF A$="H" OR A$="h" THEN RUN"AUTOPROG.BAS"
280 IF A$="Y" OR A$="y" THEN RUN"AUTOPROG.BAS" ELSE 250
290 CLS
300 PRINT TAB(T)"Automatic Programmer"
310 PRINT TAB(T)"PROGRAM PROOFER"
320 PRINT TAB(T)"By:    David D. Busch"
330 RETURN
340 CLS
350 CLOSE
360 PRINT:PRINT

365 ' *** Input filename to be proofed ***

370 LINE INPUT"ENTER FILE NAME  :  ";F$
380 FOR N=1 TO LEN(F$)
390 T=ASC(MID$(F$,N,1))
400 IF T>96 AND T<123 THEN MID$(F$,N,1)=CHR$(T-32)
410 NEXT N
420 IF LEFT$(F$, 4)="HELP" OR F$="H" THEN GOSUB 1870
430 IF LEN(F$)>12 THEN PRINT"File name too long!":PRINT:GOTO 370
440 S9=INSTR(F$,".BAS")
450 IF LEN(LEFT$(F$,S9))>8 THEN PRINT"File name too
    long!":PRINT:GOTO 370
460 IF S9=0 THEN PRINT "MUST INCLUDE .BAS EXTENSION!":GOTO 370
470 IF F$="" GOTO 370
480 RESTORE
490 DATA ABS,AND,ASC,ATN,AUTO,BEEP,BLOAD,BSAVE,CALL,CDBL
500 DATA CHAIN,CHDIR,CHR$,CINT,CIRCLE,CLEAR,CLOSE,CLS,COLOR
510 DATA COM,COMMON,CONT,COS,CSNG,CSRLIN,CVD,CVI,CVS
520 DATA "DATA",DATE$,DEF,DEFDBL,DEFINT,DEFSNG,DEFSTR
530 DATA DELETE,DIM,DRAW,EDIT,ELSE,END,ENVIRON,ENVIRONS,EOF
```

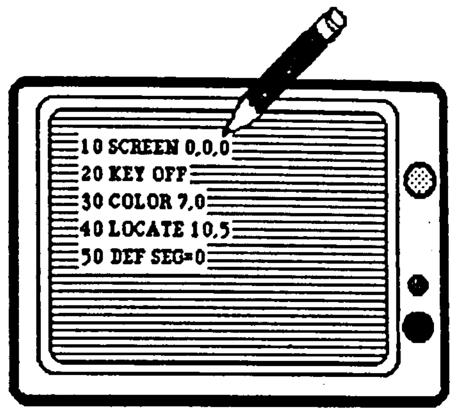
```
540 DATA EQV,ERASE,ERDEV,ERDEV$,ERL,ERR,ERROR,EXP,FIELD
550 DATA FILES,FIX,FN,FOR,FRE,GET,GOSUB,GOTO,HEX$,IF,IMP
560 DATA INKEY$,INP,INPUT,INPUT#,INPUT$,INSTR,INT,
    INTER,IOCTL,KEY
570 DATA KILL,LEFT$,LEN,LET,LINE,LIST,LLIST,LOAD,LOC,LOCATE
580 DATA LOF,LOG,LPOS,LPRINT,LSET,MERGE,MID$,MKDIR,MKD$,MKI$
590 DATA MKS$,MOD,MOTOR,NAME,NEW,NEXT,NOT,OCT$,OFF,ON,OPEN
600 DATA OPTION,OR,OUT,PAINT,PEEK,PEN,PLAY,PMAP,POINT,POKE
610 DATA POS,PRESET,PRINT,PRINT#,PSET,PUT,RANDOMIZE,READ,"REM"
620 DATA RENUM,RESET,RESTORE,RESUME,RETURN,RIGHT$,RMDIR,RND
630 DATA RSET,RUN,SAVE,SCREEN,SGN,SHEELL,SIN,SOUND,SPACE$
640 DATA SPC(,SQR,STEP,STICK,STOP,STR$,STRIG,STRING$,SWAP
650 DATA SYSTEM,TAB,TAN,THEN,TIME$,TIMER,TO,TROFF,TRON,USING
660 DATA USR,VAL,VARPTR,VARPTR$,VIEW,WAIT,WEND,WHILE,WIDTH
670 DATA WINDOW,WRITE,WRITE#,XOR
680 CLS:PRINT:PRINT
690 PRINT TAB(10)"THIS MODULE WORKS ONLY ON FILES WHICH HAVE"
700 PRINT TAB(10)"BEEN SAVED IN NON-COMPRESSED (ASCII) FORMAT"
710 PRINT TAB(10) " Use this syntax:    SAVE
    ";CHR$(34);"filename";CHR$(34)",A"
720 PRINT
730 PRINT TAB(8)"If you see garbage loading, you probably have"
740 PRINT TAB(8)"forgotten to save your file in ASCII format."
750 PRINT
760 PRINT TAB(12)" -- A  few seconds please -- "

765 ' *** Read GOOD names into array ***

770 :  FOR N=1 TO 172
780 :     READ D$
790 :     D2=ASC(LEFT$(D$, 1))-64
800 :     IF D2<>PREVIOUS THEN PREVIOUS=D2: D=1
810 :     WRD$(D2,D)=D$
820 :     D=D+1
830 :  NEXT N
840 PRINT:PRINT
850 CLS:PRINT:PRINT
860 PRINT TAB(14)" --   Reading in Program Lines -- "
870 PRINT

875 ' *** Open Program, Read in Lines ***
880 OPEN"I",1, F$
890 IF EOF(1)THEN 1310
900 LINE INPUT#1, A$
910 TEST$=""
920 PRINT A$
930 FL=0
```

```
940 SFLAG=0
950 P=INSTR(A$, CHR$(32))
960 SEG$=MID$(A$, P+1)
970 L=LEN(SEG$)+1

975 ' *** Check for keyword delimiter ***

980 :  FOR N1=1 TO L
990 :    M$=MID$(SEG$, N1, 1)
1000 :     IF SFLAG<>1 THEN 1020
1010 :     IF M$=CHR$(34)THEN 1050 ELSE 1230
1020 :     IF M$=")" OR M$="+" OR M$="-" OR M$=CHR$(32) OR M$="="
           OR M$="(" OR M$=CHR$(34)OR M$="," OR M$=":" OR M$="<"
           OR M$=">" OR M$="#" OR M$="/" OR M$="*" OR M$=CHR$(10)
           OR M$="" THEN 1050
1030 :     TEST$=TEST$+M$
1040 :     GOTO 1230
1050 :     IF SFLAG=1 THEN SFLAG=0: TEST$="": GOTO 1230
1060 :     IF M$=CHR$(34)THEN SFLAG=1:IF MID$(SEG$,
           N1-1,1)=CHR$(32)THEN TEST$=""
1070 :     IF M$="(" THEN LP=LP+1
1080 :     IF M$=")" THEN RP=RP+1
1090 :     FL=0
1100 :     IF TEST$="" THEN 1230
1110 :     IF TEST$="REM" OR TEST$="'" THEN 1240
1120 :     IF VAL(TEST$)>0 THEN TEST$="": GOTO 1230
1130 :     A=ASC(LEFT$(TEST$, 1))
1140 :     IF A<65 OR A>90 THEN TEST$="": GOTO 1230
1150 :     A=A-64
1160 :       FOR N2=1 TO 30
1170 :         IF WRD$(A,N2)="" THEN FLAG=0:N2=30: GOTO 1210
1180 :         IF TEST$=WRD$(A,N2) THEN FLAG=1: N2=30: GOTO 1210
1190 :       NEXT N2
1200 :     FLAG=0
1210 :     IF FLAG=0 THEN NU=NU+1: BAD$(NU)=TEST$+" : LINE
           "+LEFT$(A$, P)
1220 :     TEST$=""
1230 :   NEXT N1
1240 IF RP=LP THEN 1290
1250 NI=NI+1

1255 ' *** Paren missing ***

1260 PAR$(NI)="LINE " +LEFT$(A$, P)+" :  MISSING "
1270 IF RP>LP THEN P$="LEFT" ELSE P$="RIGHT"
1280 PAR$(NI)=PAR$(NI)+P$+" PARENTHESIS"
1290 RP=0:LP=0
```

```
1300 GOTO 890

1305 ' *** Display results ***

1310 CLS:PRINT:PRINT
1320 PRINT TAB(8)"Do you want output to go to printer?"
1330 GOSUB 160
1340 IF A$="Y" OR A$="y" THEN PFLAG=1
1350 GOSUB 1610
1360 ZU=1

1365 ' *** Show BAD words and Variables ***

1370 :  FOR N4=1 TO NU
1380 :    IF ZU MOD 16=0 THEN GOSUB 1570
1390 :    IF BAD$(N4)=BAD$(N4-1)THEN 1430
1400 :    PRINT BAD$(N4)
1410 :    IF PFLAG=1 THEN LPRINT BAD$(N4)
1420 :    ZU=ZU+1
1430 :  NEXT N4
1440 GOSUB 1570
1450 Z3=1

1455 ' *** Show Missing Parens ***

1460 :  FOR Z3=1 TO NI
1470 :    IF Z3 MOD 16=0 THEN GOSUB 1570
1480 :    PRINT PAR$(Z3)
1490 :    IF PFLAG=1 LPRINT PAR$(Z3)
1500 :  NEXT Z3
1510 PRINT
1520 PRINT TAB(20)" -- END OF LIST -- "
1530 PRINT
1540 PRINT TAB(15)"HIT ANY KEY TO RETURN TO MAIN MENU"
1550 GOSUB 160
1560 GOTO 340
1570 PRINT
1580 PRINT TAB(22)"HIT ANY KEY"
1590 GOSUB 160
1600 RETURN
1610 CLS:PRINT:PRINT
1620 PRINT
1630 PRINT
1640 PRINT TAB(14)"  ** POSSIBLE MISPELLINGS AND VARIABLES **"
1650 PRINT
1660 RETURN
```

```
1665 ' *** Error Trap ***

1670 IF ERR<>53 GOTO 1740
1680 CLS:PRINT
1690 PRINT TAB(20)"That file does not exist!"
1700 FOR N9=1 TO 500
1710 NEXT N9
1720 CLS
1730 RESUME 840
1740 PRINT:PRINT
1750 PRINT TAB(20)"*****  UNKNOWN ERROR  *****"
1760 PRINT TAB(25)"IN LINE ";ERL
1770 FOR N9=1 TO 500
1780 NEXT N9
1790 RESUME 340
1800 CLS:PRINT:PRINT
1810 RETURN
1820 PRINT
1830 PRINT TAB(15)"Hit any key to resume program"
1840 GOSUB 160
1850 RETURN
1860 GOSUB 1800

1865 ' *** Help Routine ***

1870 CLS:PRINT
1880 PRINT TAB(8)"Program wants the name of file to be
     proofread. Must"
1890 PRINT TAB(8)"be a legal Disk basic name, or your input
     will be"
1900 PRINT TAB(8)"rejected."
1910 PRINT
1920 LINE INPUT"ENTER FILENAME :";F$
1930 RETURN
1940 CLOSE
1950 END
```

Chapter 8



Automatic Programmer Documentation

Care to coast awhile? Here's a program you don't even have to key in. Well, that is not entirely accurate. Automatic Programmer Documentation is a help file for the preceding three modules. It is included here to demonstrate how such help programs can be used to make a complex piece of software more usable by a beginner. The program itself actually has no other function than to serve as an introduction to the Automatic Programmer series. You have four options in this case.

1. If you have purchased the disk containing all the programs in this book, the program is included on your disk. It will be called as needed by the three Automatic Programmer programs and serves as a menu gateway to them.
2. You may type in the program as presented.
3. You can type in the working program lines, but write the display lines using Screen Editor. It will prepare the screens for you with less typing on your part.
4. Just skip this chapter entirely and do without the help file when running the other three programs.

Help files are one way of making programs self-documenting. At the same time, they allow the programmer to keep the size of the main program within reasonable limits. In this instance, the Automatic Programmer programs each have some help messages built in for use when the program is running. The help file is used only at the beginning because loading AUTOPROG.BAS erases any variable values that had been established by the calling program. Going from this file back to one of the other programs initializes the variables once again.

There are several ways around this problem. One solution is to place needed values into protected locations in memory, which are not written over by new programs. You can then PEEK these values and restore them to the variables. BASIC also allows CHAINing between programs to accomplish the same thing using only BASIC keywords. A better choice might be to store each help screen in the form of a sequential file, READ in that file, and print the information to the screen. The variables that the message is read into can be used over and over with each new message, and so only a given amount of memory is taken up. This doesn't take into account variable "garbage collection," but that should be a problem only when help screens are accessed frequently, and the messages are very long.

You can see from this that professional-level programs may have as much programming time devoted to help messages and error traps as to the actual program functions themselves. Such programs are very long (and would be tedious to type in). Be thankful that this book keeps the concept to a bare minimum. You can get enough help to operate the programs successfully–but not so much that you won't be able to type them in at all.

Now, wasn't that easy? When the user specifies HELP in one of the Automatic Programmer modules, a branch to a line that reads RUN "AUTOPROG.BAS" will take place. This program will then be loaded, and display the introduction to the other programs. At the end, and INKEY$ loop will accept one of three menu choices, loading and RUNing one of the three Automatic Programmer modules. That's all there is to it. Class dismissed for recess.

Listing 8: The Automatic Programmer Documentation Program

```
10 ' *********************************
20 ' *                               *
30 ' * Auto Programmer Instructions  *
40 ' *                               *
50 ' *********************************
60 KEY OFF
70 ON KEY(10) GOSUB 1030
80 KEY(10) ON
90 SCREEN 0,0,0
100 COLOR 7,0
110 ON ERROR GOTO 250
120 CLS
130 PRINT TAB(30)"Automatic Programmer"
140 PRINT TAB(31)"By:David D. Busch"
150 PRINT:PRINT
160 PRINT TAB(8)"This program allows you to use your computer to write"
170 PRINT TAB(8)"some of the Basic program lines for many common programs"
180 PRINT TAB(8)"automatically.  It will produce a 'skeleton' coding"
190 PRINT TAB(8)"structure which you can 'flesh' out with subroutines of"
200 PRINT TAB(8)"your own.  Many initial 'housekeeping' tasks, such as"
210 PRINT TAB(8)"dimensioning an array, CLEARing memory, writing instruct-"
220 PRINT TAB(8)"ional screens (like this one), menus, are done for you."
230 PRINT
240 GOTO 290
250 IF ERR=53 GOTO 270
260 PRINT"UNKNOWN ERROR IN LINE #"ERL:FOR N=1 TO 500:NEXT N:RESUME 120
270 PRINT"PLEASE INSERT DISK CONTAINING PROPER FILES"
280 PRINT" IN DISK DRIVE":CLS:LOCATE 3,8:RESUME 850
290 PRINT TAB(8)"Data base management programs lend themselves to"
300 PRINT TAB(8)"this approach.  Automatic Programmer has a number of"
310 PRINT TAB(8)"useful functions that will save you time:"
320 PRINT
330 PRINT TAB(8)"1.) You can use it to write instructional screens."
```

```
340 PRINT TAB(8)"Instead of mapping out pages, like this one, and writing"
350 PRINT TAB(8)"program lines to reproduce the text on the screen, you"
360 PRINT TAB(8)"can enter the material exactly as you want it to appear"
370 PRINT TAB(8)"using cursor control and full-screen editing.  All alpha"
380 PRINT TAB(8)"numeric characters and symbols may be used.  Then, pro-"
390 PRINT TAB(8)"gram lines will be written and saved to disk."
400 GOSUB 960
410 PRINT TAB(8)" After a screen has been created, you may renumber it"
420 PRINT TAB(8)"so that the line numbers do not conflict with an existing"
430 PRINT TAB(8)"program, and MERGE the two.  This process may be repeated"
440 PRINT TAB(8)"to create several frames or menus for a business, computer"
450 PRINT TAB(8)"aided instruction, games, or other program."
460 PRINT
470 PRINT TAB(8)"2.) Automatic Programmer may also be used to  create"
480 PRINT TAB(8)"entire program skeletons for you to work with.  The"
490 PRINT TAB(8)"'screen' writer module may be used, along with several"
500 PRINT TAB(8)"others.  It will write program lines to dimension a"
510 PRINT TAB(8)"string array, build disk I/O routines to fill an array"
520 PRINT TAB(8)"and dump its contents to a disk file."
530 PRINT
540 PRINT TAB(8)" If your program will use DATA lines, you may simply"
550 PRINT TAB(8)"enter the actual data itself.  Automatic Programmer will"
560 PRINT TAB(8)"insert line numbers, DATA statements, and write a routine"
570 PRINT TAB(8)"to READ that data into an array for later manipulation."
580 GOSUB 960
590 PRINT TAB(8)" You may construct a menu, too.  If you choose to"
600 PRINT TAB(8)"build a custom menu, you can make use of the screen"
610 PRINT TAB(8)"writer routine.  The computer can also build a menu for"
620 PRINT TAB(8)"you, from your input of the number of choices, labels for"
630 PRINT TAB(8)"those choices, and other data."
640 PRINT TAB(8)" When using this feature, the program will write ON...  "
650 PRINT TAB(8)"GOSUB lines for you, and insert REMARK pointers at those"
660 PRINT TAB(8)"locations so you know where to write each subroutine.  "
670 PRINT
680 PRINT TAB(8)" The program lines written include error traps and"
690 PRINT TAB(8)"other helpful features that you do not have to program"
700 PRINT TAB(8)"yourself.  Although Automatic Programmer will not write"
710 PRINT TAB(8)"a complete program, it will get the basics out of the"
720 PRINT TAB(8)"way fast, and allow you to use your creativity where it"
730 PRINT TAB(8)"counts the most."
740 PRINT
750 PRINT TAB(8)"3.) Automatic Programmer can also be used, to a"
760 PRINT TAB(8)"limited extent, to proofread the programs you have writ-"
770 PRINT TAB(8)"ten.  It will check for misspelled keywords, mismatched"
780 PRINT TAB(8)"parentheses, and some other errors.  "
790 GOSUB 960
800 LOCATE 8,8
810 PRINT "Please note:"
820 PRINT
830 PRINT TAB(8)"o Program to be proofed must be saved in ASCII form."
840 GOSUB 960
850 PRINT TAB(18)"Hit 'R' to repeat instructions."
860 PRINT TAB(18)" == Press ==="
870 PRINT TAB(18)"1.) To run Screen Editor"
880 PRINT TAB(18)"2.) To run DB Starter"
890 PRINT TAB(18)"3.) To run Program Proofer"
900 A$=INKEY$:IF A$="" GOTO 900
```

```
910 IF A$="R" OR A$="r"GOTO 120
920 IF A$="1" THEN RUN"SCREEN.BAS"
930 IF A$="2" THEN RUN "DBSTART.BAS"
940 IF A$="3" THEN RUN "PROOFER.BAS"
950 GOTO 900
960 LOCATE 25,12
970 COLOR 0,7
980 PRINT "-- HIT ANY KEY FOR MORE, F10 TO SKIP INSTRUCTIONS --";
990 COLOR 7,0
1000 IF INKEY$="" GOTO 1000
1010 CLS:LOCATE 3,1
1020 RETURN
1030 CLS:LOCATE 8,1
1040 RETURN 860
```

Chapter 9

Global Replacer

So far, you've seen that the key to teaching your IBM PC to program itself has been to provide it with a simple set of instructions that it can follow to do what you want. Many times, these are almost trivial, repetitious tasks that the computer can do much faster than we can. For example, a human could easily go through a program looking for REMarks, and deleting them manually. We, however, might overlook one or two. And, even with the IBM PC's screen editor, moving the cursor around and pressing DEL or BACKSPACE repeatedly is time consuming and a bit boring. With programs like REM-over, we have been able to command the IBM PC to do this task for us.

So called *global* search and replace is another feature that can automate a time consuming or error prone task. With this capability you can find every instance of a string, and if you wish, change it to something else.

Global search and replace is a strong feature of microcomputer word processing programs. All WP programs for the IBM PC have this capability, which allows the user to search through a text file and change all occurrences of one string to another. If you wrote PRINT and you meant LPRINT, the change will take just a few seconds.

MAKING CHANGES WITHOUT A WORD PROCESSING PROGRAM

What if you want to do the same function not on a text file, but a program file? Some word processing programs will load an ASCII format program, allow text manipulation, and then save the new program, again in ASCII; however, not all WP software allows this. Many do not let you choose which instances to replace (they are always global). That is, you may have the choice of replacing ALL occurrences, or of searching to each spot and then manually typing in the replacement string. Some of us do not have word processing programs in any case, either because we don't use our PCs for word processing, haven't gotten around to buying a WP program, or can't justify the cost of one.

Here is the solution to your problems. It is another program in the "REM-over" mold. This one, Global Replacer, demonstrates how one program can be adapted to perform a second function. In concept, the two are almost identical. The difference is instead of searching for remarks and then deleting them, the program looks for ANY string of the operator's choice. Then the string is replaced with a second string.

Unlike some word processing programs, however, the user is shown each occurrence of the search string and offered the opportunity to replace it. You can pick and chose which to replace and which to leave alone.

The search string is input into S$ in line 230. (Figure 9-1 shows the variables used in Global Replacer.) Since LINE INPUT is used, the string can contain commas and other string delimiters. The replacement string is entered into RE$, in line 260. Then the input and output files are opened, and the first program (or text) line loaded into A$, in line 360.

The user has been offered the option of whether or not the program queries before making the replacement. A search routine, which is basically identical to that used in REM-over, hunts for the string. The difference is that in line 390, where the former program had R=INSTR(P,A$,"REM"), Global substitutes S$ for REM. If R does not equal zero, then the string searched for has been successfully located. At that point, the program line is cut apart into two sections. L$ stores everything in the line up to the beginning of the search string. R$ includes the rest of the line AFTER the search string. Another string, Y$, which is a series of blanks of the same length as the replacement string, is constructed.

If the user has specified querying, control goes to line 460, where an INKEY$ loop awaits keyboard input. Each time through the loop, L$, Y$, and R$ are printed on the same line; then there is a short delay, and L$, RE$, and R$ are printed. The result is a flashing display with the left and right portions of the program line remaining on the screen, while the potential replacement flashes on and off in its place. A **Replace it?** prompt asks for a decision. The program will only replace the string if a Y is entered. Any other key will leave the program line as it was.

Once the string has been replaced, the program branches back to search the rest of the line. If the

A$	Stores program line being searched.
B$	Used in INKEY$ loop.
CH$	Used in INKEY$ loop.
E	Length of string being searched for.
F$	File name of program being searched.
F1$	Name of output file.
L$	Left portion of program line.
N1	Loop counter.
P	Position to begin search.
R	Position of searched for string.
RE$	Replacement string.
S$	String to search for.
Y$	String of spaces as long as string replacing with.

Fig. 9-1. Variables used in Global.

arch string is not found, the program line is inted to the disk in line 700, and a new program e fetched.

DVANTAGES OF GLOBAL REPLACER

Global Replacer is a short but powerful program that will let you make changes rapidly in a given program. Should you decide to change the ame of a variable, substitute one keyword for another (e.g., LPRINT for PRINT), or do some anges of prompts and other material within otation marks, it will handle them all. Its chief advantage over using a text editor for the same chore is the ability to examine each line before making the change. In addition, those without word processing programs can use this utility.

As always, you can abort this program by pressing F10. Your original file will not be harmed—nothing is done to it, in any case. GLOBAL, like most of the other programs in this book, only reads in the original file and writes an entirely new file with the changes to disk. The source file is untouched, and thus aborting the program has no effect on it.

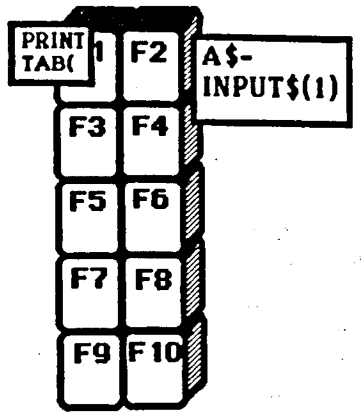
Listing 9: The Global Replacer Program

```
10 ' ****************
20 ' *              *
30 ' *   GLOBAL     *
40 ' *              *
50 ' ****************

55 ' *** Initialize ***

60 KEY OFF
70 SCREEN 0,0,0
80 ON KEY(10) GOSUB 790
90 KEY(10) ON
100 COLOR 7,0
110 CLS:PRINT:PRINT
120 LOCATE 25,30
130 COLOR 16,7
140 PRINT" Hit F10 to abort. ";
150 COLOR 7,0
160 LOCATE 4,20

165 ' *** Enter names of files ***

170 PRINT "Enter name of program to be processed :"
180 LINE INPUT F$
190 PRINT TAB(26)"Enter name of output file :"
200 LINE INPUT F1$
210 CLS:PRINT:PRINT
220 PRINT TAB(26)"Enter string to search for :"
230 LINE INPUT S$
240 CLS:PRINT:PRINT
```

```
250 PRINT TAB(25)"Enter string to replace with :"
260 LINE INPUT RE$
270 CLS:PRINT:PRINT
280 PRINT TAB(17)"Do you want to choose whether to replace
    each?"
290 PRINT TAB(37)"(Y/N)"
300 CH$=INKEY$:IF CH$="" GOTO 300
310 IF CH$="Y" OR CH$="y" THEN CH=1
320 CLS

325 ' *** Open Disk Files ***

330 OPEN "I",1,F$
340 OPEN "O",2,F1$
350 IF EOF(1) GOTO 730

355 ' *** Load a line ***

360 LINE INPUT #1,A$
370 IF CH=1 THEN CLS
380 P=1
390 R=INSTR(P,A$,S$)
400 IF R=0 GOTO 700
410 L$=LEFT$(A$,R-1)
420 E=LEN(S$)
430 R$=MID$(A$,R+E)
440 Y$=STRING$(LEN(RE$),32)
450 IF CH=0 THEN GOTO 670

455 ' *** Replace it? ***

460 B$=INKEY$
470 LOCATE 3,4
480 PRINT L$;
490 COLOR 0,7
500 PRINT Y$;
510 COLOR 7,0
520 PRINT R$
530 FOR N1=1 TO 50:NEXT
540 LOCATE 3,4
550 PRINT L$;
560 COLOR 0,7
570 PRINT RE$;
580 COLOR 7,0
590 PRINT R$
600 FOR N1=1 TO 50:NEXT
```

```
620 PRINT "Replace it? (Y/N)"
630 IF B$="" GOTO 460
640 IF B$="Y" OR B$="y" GOTO 670
650 P=INSTR(P,A$,S$)+LEN(S$)-1
660 GOTO 390
670 A$=L$+RE$+R$
680 P=INSTR(P,A$,RE$)+LEN(RE$)-1
690 GOTO 390

695 ' *** Print to disk ***

700 PRINT #2,A$
710 IF CH=0 THEN PRINT A$
720 GOTO 350
730 CLOSE

735 ' *** Do it again? ***

740 PRINT:PRINT
750 PRINT TAB(29)"Process another file?"
760 PRINT TAB(37)"(Y/N)"
770 A$=INKEY$:IF A$="" GOTO 770
780 IF A$="Y" OR A$="y" THEN RUN
790 CLOSE
800 CLS
810 END
```

Chapter 10



Key Definer

A long time ago, in a galaxy far, far away, microcomputers didn't have function keys. Some didn't even have cursor keys! All users had available were the standard alphanumerics and, if they were lucky, a control key, an escape key, and a few others. This lack of available extra keys led to some interesting programming solutions. WordStar, which could be used on computers without cursor keys, required strange combinations of control-key plus another key to move the cursor around on the screen. Some commands called for two and three key combinations.

Even more interesting, one popular word processing program for the Tandy line asked the user to think of the @ key as a control key. Of course, that meant that there was no way to print the @ character—except that the programmer "moved" it to Shift-0 (shift-zero). Without a CAPS LOCK key, this same WP program used Shift-@. And so it went.

One of the nicest features of the latest generation of microcomputers, like the IBM PC and PCjr, is that they include lots of extra keys. In addition to 10 function keys, there are control keys, including ALT, INS, DEL, NUM LOCK, BREAK, ESC, and others.

These keys can be used to make programming easier and programs easier to use. Function keys are used in two ways. First, you can write the program with an ON KEY(n) GOSUB interrupt, so that when the designated key is pressed, control goes to the desired subroutine. Or, you may actually want the key to return a set of characters when it is pressed. The IBM PC boots up set to return strings like LIST, LOAD, SAVE, and so forth when a function key is hit.

You may sniff that, of course, it is a simple matter to write a program so that hitting a function key, like F1, will take the user to a desired subroutine in a flash. That has been done repeatedly in this book. But, you continue, there's not a lot of use in having LIST or SCREEN 0,0,0 available to the programmer at the touch of a key.

Well, you should know by now that if you don't like the way your Automatic IBM is treating you—change it! Key Definer is a short program that will write an even shorter program that redefines all 10 special function keys for you automatically.

USING KEY DEFINER

There are two ways to use Key Definer. First, you can run the program from BASIC, and enter the new key definitions you'd like. When you're finished, hit F1. As if by magic, Key Definer will be gone from memory, and your new key definitions will be implemented. What's more, there will be a new disk file that you can run anytime you like to summon those same key definitions.

That disk file is the second way to use this program. It can compile a selection of different function key settings—as many as you want—that you can load at your command. Or, you can have those definitions loaded automatically by means of a "custom" DOS command you have created. (Custom DOS commands will be explained in Chapter 18.) In this mode, you could type BASIC23 from DOS, and have Key Definition File #23 activated. Or, you might want to put this line in your AUTOEXEC.BAT file:

```
BASIC KEYDEF.BAS
```

Every time your PC is booted, it will go to BASIC with the key definitions in the specified file name (I used KEYDEF.BAS here) loaded automatically. We'll explore this aspect later. You can enjoy Key Definer right now!

REASONS FOR REDEFINING FUNCTION KEYS

Exactly how and why would you redefine the special function keys of your PC? Those who think that special function keys are best applied as a kind of shorthand to eliminate typing in GOTO or other phrases suffer from a failure of imagination. The nice thing about general-purpose microcomputers like the IBM is that they can be custom-configured to perform specialized tasks tailored to the exact needs of the end user. Thanks to the sophistication of DOS 2.0 and beyond, patches, special ROM cartridges, and utility programs, many features can be available on power-up, or, at most, at the press of a few keys.

User-programmable special function keys can do a great deal more than print out a lengthy BASIC keyword. Here are some applications you might not have thought of.

Program a key so that F1 produces FILES, or SYSTEM or some other command you use frequently. Your function keys can store a string of up to 15 characters, enabling you to redefine them to include lines you frequently use in programs, such as A$=INPUT$(1), or OPEN "O",1,F$.

User-programmable keys are truly the programmer's friend. Do you frequently renumber your programs during writing to make additional room between lines? Program a key to yield RENUM 10,10 whenever you strike it. Set another key to PRINT TIME$. Then, hit that function key to see the correct time anytime you want. Your uses are limited only by the number of keys available for programming.

HOW TO REDEFINE FUNCTION KEYS

The correct syntax for redefining the PC's function keys is as follows:

```
KEY n,string
```

For example:

```
KEY 1,"RENUM 10,10"
KEY 10,"FILES"
```

You do not need to activate these function keys with the KEY ON statement. Once defined, they are instantly ready for your use while programming in command mode. You can turn off the display of the key definitions in line 25 by entering KEY OFF, and turn it back on again with KEY ON. That affects only the display. Between times, pressing F1 will still produce the string defined for that key. To truly turn it off, you need to define the key as a null string:

```
KEY 1,""
KEY 10,""
```

Don't confuse the strings produced by pressing a special function key with the ON KEY(n) GOSUB feature. That is entirely different. You can have redefined keys (useful from command mode) and ON KEY(n) routines (useful in your programs) at the same time, with different results.

Here is the main difference: when ON KEY(n) is activated, statements like LINE INPUT will ignore the function key's string, but still recognize that the function key has been pressed. Assume you have redefined F1 to equal "RENUM 10,10". If you ran the following program line:

```
10 LINE INPUT A$
```

and pressed F1, followed by the Enter key, then RENUM 10,10 would be printed to the screen, and A$ would equal "RENUM 10,10".

However, add these three lines:

```
5 ON KEY(1) GOSUB 100:KEY(1) ON
50 STOP
100 PRINT "YOU PRESSED F1!"
```

Now, when line 10 is run, if you press F1, nothing will appear on the screen. Neither will BASIC branch to line 100. Instead, it will wait until you press the Enter key (just in case you want to enter something into the LINE INPUT) and then immediately jump to line 100. A$ will not contain "RENUM 10,10." So, our redefined keys do not interfere with ON KEY(n). You've also learned, however, that LINE INPUT won't let you jump immediately to the subroutine you want to interrupt with. For that reason, programs using function key interrupts in this book that require LINE INPUT-type entries (that is, commas and other delimiters must be acceptable) use INKEY$ and concatenation.

You now know enough about the IBM's function keys to know that Key Definer is a very simple program. Figure 10-1 shows the variables used.

A string array is set up in line 60 to hold 10 key definitions, one for each of the 10 special function keys. Then F1, used to end the input session, is activated.

You are asked which key to define. Here I use the A$=INKEY$ technique mentioned, so that F1 can, indeed, interrupt the entry when we are finished.

The key to be defined, K, will be given your desired string, D$. Your definition is checked to make sure that it is 15 characters or less. You can enter nothing, to cancel out a key completely, if you wish.

A counter, CU, keeps track of the number of keys defined, and the K element of the array D$(n) is loaded with your chosen string. This process repeats as many times as you want until F1 is pressed. You may define any or all of the 10 function keys, redefine some, skip some, or any combination.

When F1 is pressed, the program branches to line 520, where a file, KEYDEF.BAS is opened. A FOR-NEXT loop from 1 to 10 writes your definitions to the disk. If you have not defined a key, a null definition is written. Note: this will cancel out any default definitions for those keys.

Variable C corresponds to the line number in the new short program being created. The first line number will always be one. If you have defined 10 keys, then 10 line numbers will be used. The pro-

K	Key to be redefined.
K$	New string to assign to that key.
N	Loop counter.

Fig. 10-1. Variables used in Key Definer.

gram line is built from the line number, C, plus "KEY" + STR$(N) + " , " + CHR$(34) + D$(N) + CHR$(34). This produces a line like:

```
1 KEY 5,"RENUM 10,10"
```

The final step is to write one more line, containing the command NEW. Then the new program just created, KEYDEF.BAS will be run. It will redefine your keys and then erase itself from memory when it encounters the NEW command. The program KEYDEF.BAS, however, still resides on your disk and can be used later if you wish. RENAME it under some other file name so that subsequent runs of Key Definer won't write over the existing file with the new one.

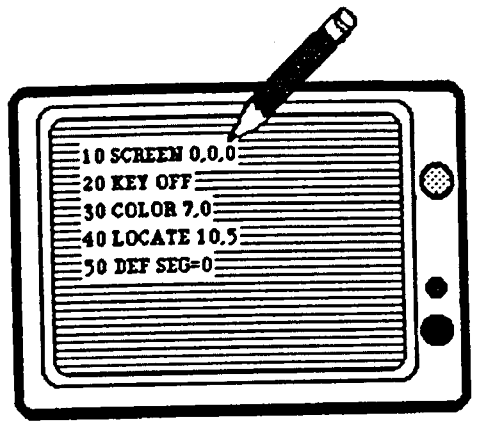
Listing 10: The Key Definer Program

```
10 ' *****************
20 ' *               *
30 ' *   KEY DEFINER *
40 ' *               *
50 ' *****************

55 ' *** Initialize ***

60 DIM D$(15)
70 KEY OFF
80 SCREEN 0,0,0
90 COLOR 7,0
100 ON KEY(1) GOSUB 520
110 ON KEY(10) GOSUB 680
120 KEY(1) ON
130 KEY(10) ON
140 CLS:PRINT:PRINT
150 K$=""
160 PRINT TAB(26)"Which key to define (1-10)?"
170 PRINT
180 PRINT TAB(33)"Hit F1 to finish definitions."
190 PRINT TAB(33)"Hit F10 to abort and end program"

195 ' *** Enter key to be defined ***

200 D$=""
210 LOCATE 12,5:COLOR 0,7:PRINT" DEFINE KEY # ";:
    COLOR 7,0:PRINT K$
220 A$=INKEY$:IF A$="" GOTO 220
230 IF A$=CHR$(13) THEN GOTO 270
240 IF A$<"0" OR A$>"9" GOTO 220
250 K$=K$+A$
260 GOTO 210
270 K=VAL(K$)
280 IF K<1 OR K>10 THEN GOTO 140
```

```
285 ' *** Enter definition ***

290 LOCATE 12,27:PRINT"Enter definition for key
    F";MID$(STR$(K),2);", then [ENTER]"
300 LOCATE 14,5:PRINT SPACE$(20);
310 A$=INKEY$:IF A$="" GOTO 310
320 IF A$=CHR$(8) AND D$<>"" THEN
    D$=LEFT$(D$,LEN(D$)-1):LOCATE 14,5:PRINT SPACE$(20):
    LOCATE 14,5:GOTO 360 ELSE IF A$=CHR$(8) AND D$=""
    THEN GOTO 360
330 D$=D$+A$
340 IF A$=CHR$(13) GOTO 420
350 IF LEN(D$)>15 THEN BEEP:LOCATE 25,4:PRINT
    SPACE$(70);:COLOR 0,7:LOCATE 25,4:PRINT"ONLY 15 CHARACTERS
    PLEASE!!";:FOR N=1 TO 1000:NEXT N:LOCATE 25,4:COLOR
    7,0:PRINT SPACE$(50);:D$="":GOTO 290
360 LOCATE 14,5
370 PRINT D$
380 LOCATE 25,4
390 COLOR 0,7
400 PRINT " LENGTH OF STRING : ";:COLOR 7,0:LOCATE 25,29:PRINT
    LEN(D$);:COLOR 0,7:LOCATE 25,40:PRINT" LIMIT 15 ";:
    COLOR 7,0
410 GOTO 310
420 CU=CU+1
430 D$=LEFT$(D$,LEN(D$)-1)

435 ' *** Append C/R ? ***

440 LOCATE 25,4
450 PRINT"End with carriage return?  (Y/N)";
460 A$=INKEY$:IF A$="" THEN GOTO 460
470 IF A$="Y" OR A$="y" THEN GOTO 480 ELSE M$="":GOTO 500
480 IF LEN(D$)=15 THEN BEEP:LOCATE 25,4:PRINT
    SPACE$(55);:LOCATE 25,4:PRINT"Sorry, too long for C/R.
    Re-enter.";:FOR N=1 TO 1000:NEXT N:LOCATE 25,4:PRINT
    SPACE$(40);:D$="":GOTO 290
490 M$=CHR$(13)
500 D$(K)=D$+M$
510 GOTO 140

515 ' *** Write file to disk ***

520 OPEN "O",1,"KEYDEF.BAS"
530 C=C+1
540 L$=STR$(C)+" KEY ON"
550 PRINT #1,L$
```

```
560 FOR N=1 TO 10
570 IF D$(N)="" GOTO 630
580 C=C+1
590 M$=""
600 IF RIGHT$(D$(N),1)=CHR$(13) THEN
    M$="+CHR$(13)":D$(N)=LEFT$(D$(N),LEN(D$(N))-1)
610 L$=STR$(C)+"  KEY"+STR$(N)+","+CHR$(34)+D$(N)+CHR$(34)+M$
620 PRINT #1,L$
630 NEXT N
640 C=C+1
650 PRINT #1,STR$(C)+"NEW"
660 CLOSE 1
670 RUN "KEYDEF.BAS"
680 CLOSE:END
```

Chapter 11



```
10 SCREEN 0,0,0
20 KEY OFF
30 COLOR 7,0
40 LOCATE 10,5
50 DEF SEG=0
```

Lister

Hardcopy program listings are a necessary evil byproduct of programming. You can't RUN a listing. If you find one in a magazine, you have to type it in, spend hours debugging it, and then cross your fingers and hope the typesetters didn't make a mistake. (It is for that reason that the programs in this book were reproduced directly from printouts from working, tested programs.)

You can't change a hardcopy listing. If you decide to make a change in a program, it's necessary to do that with the actual computer, and then printout a whole new listing.

USES OF HARDCOPY LISTINGS

So, why do we have these hardcopy printouts? Well, for one thing, a listing is less costly to reproduce than the program on some other medium. You'll find a dozen and a half programs in this book, along with witty documentation and tutorial explanation for less than $1 per program. The disks used to contain the text and programs prior to publication cost a bit more than that. A listing, which can be duplicated for a few cents a page, is an economical way of distributing a program in a form that the user can eventually transport to his or her microcomputer.

Listings are a fairly universal medium of exchange, as well. You can type some of these programs into nonIBM computers using similar BASICs, but incompatible disk formats.

For the programmer, a listing can be a debugging tool as well. There, laid out in its entirety, is the full program. It is possible to jump back and forth between subroutines much faster by using your eyes than by typing LIST 100-300 at the keyboard. Also you can view several subroutines at once, which may be difficult on the screen, if they are long or in different parts of the program.

As I said, hardcopy listings are a necessary evil. Our job is to make them slightly less evil, if possible. The way that you can do this is by formatting the listings to be a bit more readable, neater, and easier to understand.

A$	Stores program line being listed.
C$	Used in INKEY$ loop.
COL$	Width of printout.
L$	Name of file to be listed.
LL	Lines listed.
N	Loop counter.
P	Page number.
PG	Lines per page.
R$	Middle string of line being listed.

Fig. 11-1. Variables used in Lister.

PRODUCING HARDCOPY LISTINGS WITHOUT A WP PROGRAM

You can, if you wish, use many word processing programs to format and print out your listings. I present Lister, which will do the job from BASIC, for those who do not have word processing programs.

Lister combines some of the features of programs introduced previously. Like many, it loads a program and looks at each line. Then it examines the contents and performs some small trick that we programmers will find of value. In this case, it will

```
10 ' *********************
20 ' *                   *
30 ' *    Word Counter   *
40 ' *                   *
50 ' *********************
60 CLEAR 4000
70 DEFINT A-Z
80 CLS:PRINT:PRINT
90 PRINT TAB(21)"Writer's Word Counter
   "
100 PRINT
110 PRINT TAB(6)"This program will
     count the number of actual words
     in a "
120 PRINT TAB(2)"text file, or any
     file that has been stored to disk
     in ASCII "
130 PRINT TAB(2)"format.  In addition,
     it also provides the total number
     of "
140 PRINT TAB(2)"'standard ' five
     character words, and the average
     character "
150 PRINT TAB(2)"length of the words
     in the text. "
160 PRINT:PRINT TAB(17)"== Hit any
     key to continue == "
170 IF INKEY$="" GOTO 170
180 CLS:PRINT:PRINT' *** Access
     Disk File ***
```

Fig. 11-2. An example of a listing produced by Lister.

format program listings into neat, paged groups.

The program asks the user to enter the name of the file to be listed on the lineprinter. The number of columns wide is entered, along with the number of lines per page. Then the file is opened and a line input into A$. (A list of variables used in Lister is shown in Fig. 11-1.)

The program then commences a FOR-NEXT loop that begins 10 characters to the left of the desired column width. That is, if 50 columns are desired, the program starts checking a line to be listed at the 40th character. This is considered the "hot" zone. At this point, the program begins looking for either a colon or a space. When one is found, it splits the program line at the colon or space, and LPRINTs the two parts, with some spaces added to indent the second portion of the line past the line number above. The counter for the number of lines printed so far, LL, is also incremented. Whenever LL is greater than the number of lines desired per page, a new page is started, with an appropriate heading.

Note: because some computer setups hang up when attempts are made to LLIST without a printer being switched on or connected, leave the REMs shown in place while typing and debugging Lister. When everything is working fine, remove them, and your listing will go to the printer as well as to the screen. Figure 11-2 shows an example of another program in this book that has been LLISTed using Lister.

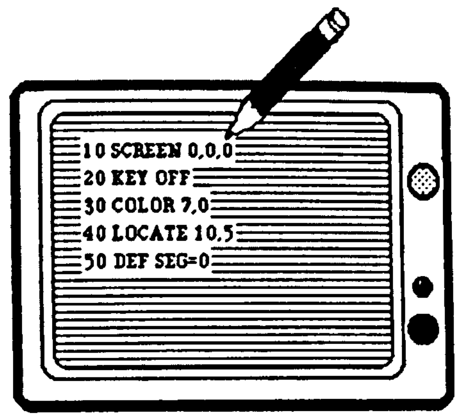
Listing 11: The Lister Program

```
10 ' ***********
20 ' *         *
30 ' *  Lister *
40 ' *         *
50 ' ***********

55 ' *** Initialize ***

60 KEY OFF
70 ON KEY(10) GOSUB 570
80 KEY(10) ON
90 SCREEN 0,0,0
100 COLOR 7,0
110 CLS:PRINT:PRINT
120 LOCATE 25,4
130 COLOR 16,7
140 PRINT " Hit F10 to abort ";

145 ' *** Enter filename ***

150 COLOR 7,0
160 LOCATE 8,24
170 PRINT "Enter name of file to be listed:"
180 LINE INPUT L$
190 PRINT TAB(29)"How many columns wide?"
200 INPUT COL$
```

```
210 COL=VAL(COL$)
220 PRINT TAB(28)"How many lines per page?"
230 INPUT PG$
240 PG=VAL(PG$)
250 P=1
260 GOSUB 500

265 ' *** Open Disk File ***

270 OPEN "I",1,L$
280 IF EOF(1) GOTO 440
290 IF LL>PG THEN GOSUB 500

295 ' *** Look For Space or Colon ***

300 LINE INPUT#1,A$
310 :  FOR N=COL-10 TO COL
320 :    R$=MID$(A$,N,1)
330 :    IF R$=CHR$(32) GOTO 380
340 :    IF R$=":" GOTO 380
350 :  NEXT N
360 LPRINT A$
370 GOTO 280
380 L$=LEFT$(A$,N)
390 LPRINT L$:LL=LL+1
400 LPRINT STRING$(5,32);
410 A$=MID$(A$,N+1)
420 IF A$="" GOTO190
430 GOTO 310
440 CLOSE

445 ' *** Do it again ? ***

450 PRINT:PRINT
460 PRINT TAB(31)"List another file?"
470 PRINT TAB(37)"(Y/N)"
480 A$=INKEY$:IF A$="" GOTO 480
490 IF A$="Y" OR A$="y" THEN RUN ELSE END

495 ' *** Page Routine ***

500 LPRINT:LPRINT:LPRINT:LPRINT
510 PRINT:PRINT:PRINT"Please insert another page."
520 C$=INKEY$:IF C$="" GOTO 520
530 LPRINT L$;" Listing Page ";P
540 LL=0
550 P=P+1
```

```
560 RETURN

565 ' *** Abort ***

570 CLOSE
580 CLS
590 END
```

Chapter 12



Translator

Most of the BASIC language's limitations stem from its original purpose as a high-level language that would be easy for beginners to learn and use. Its strongest point—the simple English keywords—provides an artificial barrier for those whose primary language is not English. Some of the largest Spanish-speaking communities in the world, for example, are in the United States. The availability of a BASIC in Spanish might make it easier for these citizens to use computers at an earlier age.

A machine language Spanish-Basic interpreter would be ideal. Programs could be written in a Hispanic version of BASIC, run, tested, and debugged in that form. Unfortunately, that would be a major undertaking, best tackled by a software house with some hopes of recouping the time investment through sales. But one-tenth of a loaf is often better than none. Translator is a simple pseudo-compiler that converts programs written in Spanish Tiny BASIC to standard BASIC for running.

Most readers will not remember Tiny BASIC, which was a very small version of BASIC used on some early microcomputers because it could be fit in an 8K ROM. It lacked many features we now consider standard in an advanced language like that available from the IBM PC.

The Translator program displays all the commands, statements, and functions available; this display can be summoned by entering HELP (or AYUDA) while the program is running.

The Translator program allows the user program to write the source code using Spanish keywords, instead of the English Basic equivalent. As each line is entered, the program checks it for various criteria (each must begin with a line number, and no more than one statement is allowed per line) and generates a new line of code, replacing each of the Spanish keywords with the English equivalent. Both versions may be saved to disk or listed at any time. Figure 12-1 provides an example of Spanish and English versions of a program.

```
Spanish Version

10 IMPRIMA "PROGRAMMA"
20 ENTRE "SU NOMBRE :";A$
30 SI A$="DAVID" LUEGO IMPRIMA "HOLA DAVID!"
40 SI A$<>"DAVID" VAYA SUB 100
50 FIN
100 IMPRIMA "HOLA,";A$
110 RETORNE

English Version

10 PRINT   "PROGRAMMA"
20 INPUT "SU NOMBRE :";A$
30 IF A$="DAVID" THEN  PRINT   "HOLA DAVID!"
40 IF A$<>"DAVID" GOSUB    100
50 END
100 PRINT   "HOLA,";A$
110 RETURN
```

Fig. 12-1. An example of a program produced by Translator.

Translator combines some of the features of Global Replacer and Program Proofer. It compares its internal list of allowable keywords with those in the input lines, and replaces them with the equivalents as needed. Figure 12-2 lists the variables used in Translator.

Editing is accomplished by reentering the line. The English ("compiled") version of the program is "object code" that may be loaded and run under your BASIC interpreter, like any BASIC program, as long as the code entered in Spanish conformed to the normal syntax rules of BASIC.

Ideally, the program should be used by a person who already knows standard BASIC to teach a Spanish-speaking person how to program.

The Spanish words chosen are not necessarily the best possible equivalents for BASIC keywords replaced. The BASIC translations were chosen using two criteria. The Spanish words had to be short and mean approximately what the BASIC equivalents mean. Because keywords are, in effect, commands, the imperative form of the verbs were used. Second, programming was made easier by selecting Spanish words that were either the same length or longer than the BASIC keywords.

To use the program, the student types RUN, in English, and is shown a summary of the commands and statements available. This list can be summoned at any time by typing HELP or AYUDA at the > prompt. An existing program may be loaded from the disk using the CARGE command. Prompts ask for the name of the program in Spanish and English. Then the program can be edited or new lines added.

At any time a specific line in Spanish can be seen by entering ALISTE xxx, where xxx is the line number. By typing ALISTE, the entire program will be presented a section at a time. Entering LIST, in English, will display the compiled English version. NEUVO (NEW) or CORRA (RUN) will erase

A$(n)	Difference in length of keywords.
A$	Line entered by user.
A1$	Used in INKEY$ loop.
B	Position of quote in line input.
C	Position of colon in line input.
COM$	Command entered by user.
CP$(n)	Array storing program lines in English.
CU	Counter.
E2$(n)	Array storing program lines in Spanish.
F$	File name.
F3$	File name.
FLAG	Shows whether instructions have been displayed.
G	Loop counter.
IG$	Program line input by user.
L	Length of program line.
N	Loop counter.
NE$	Name of program in Spanish.
NI$	Name of program in English.

Fig. 12-2. Variables used in Translator.

the current program in memory and allow the user to start over.

Only line numbers between 1 and 200 may be used, and only single statements are allowed per line. Spaces must be used after line numbers and between words. It is permissible to end a line with a space, as one is added automatically. Spaces are essential, because in searching for keywords, the program looks not for, say, the letters SI, but for <space>SI<space>. Otherwise, by the time the loop that searches for keywords got to SIGUIENTE, the word would have been changed to IFGUIENTE.

Actual translation from Spanish to English is simple. The programmer enters a line, loaded into A$ in line 1350. The entry is changed to all uppercase letters. Then the first four characters are checked to see if any of the allowable commands are included. If not, the line must begin with a line number, or else an error message is generated. A check is made for a colon outside quotation marks, which would indicate a multiple statement line. An error trap also checks to make sure that the line number is within the range allowed.

A FOR-NEXT loop beginning at line 1740 compares each word in the line with the permissible keywords, and if one is found, the equivalent English keyword is substituted for the Spanish keyword. Several subroutines take care of LISTing the program lines, stored in two string arrays. The program keeps track of the high line number used so far, in variable HIGH.NUMBER, and only goes to that value when LISTing. In that way, a lot of time is not wasted trying to LIST program lines that do not exist.

The only hitch in Translator is a problem common to all compilers. The programmer cannot run the program to test it until it has been compiled. Then if bugs are found, the compiled version cannot be changed because, in this case, the Spanish speaking person supposedly cannot understand the BASIC object code. Of course, an English-speaking person can edit it, but for those for whom Translator was intended, the object code may mean about as much as a machine language dump.

Because Translator was meant as a learning tool, it was designed to be easy to change. Keywords can be added by appending them to the proper locations in the DATA lines and adding numeric DATA that shows the difference in length between the longer Spanish keyword and the shorter English equivalent. The variable NUMBER.WORDS must also be changed to reflect the new number of words.

This program will compile from any language. The user could select keywords in, say, French, and enter them with their English BASIC counterparts in the DATA lines. All the prompts in Spanish will have to be changed as well, but these have purposely been kept to a minimum in the program.

Listing 12: The Translator Program

```
10 ' ***************
20 ' *             *
30 ' *  Translator *
40 ' *             *
50 ' ***************

55 ' *** Initialize ***

60 KEY OFF
70 ON KEY(10) GOSUB 2370
80 KEY (10) ON
90 HIGH.NUMBER=200
100 GOTO 180
110 LOCATE 25,19
120 COLOR 16,7
130 PRINT" == Hit any key == ";
140 IF INKEY$="" GOTO 140
150 COLOR 7,0
160 CLS
170 RETURN
180 DEFINT A-Z
190 SCREEN 0,0,0
200 COLOR 7,0
210 KEY OFF
220 NUMBER.WORDS=21
230 L2=200
240 C1$=CHR$(34)
250 C2$=CHR$(58)
260 C3$=CHR$(32)
270 DIM A(21), E$(21), E2$(200), CP$(200), E3$(21), SPAN$(21)
280 CLS
290 RESTORE

295 ' *** Null arrays ***

300 :  FOR N=1 TO 200
310 :    E2$(N)=""
```

```
320 :    CP$(N)=""
330 :  NEXT N

335 ' *** Read Difference Data ***

340 :  FOR N=1 TO NUMBER.WORDS
350 :    READ A(N)
360 :  NEXT N

365 ' *** Read Spanish and English keywords ***

370 :  FOR N=1 TO NUMBER.WORDS
380 :    READ E3$(N)
390 :    E3$(N)=C3$+E3$(N)+C3$
400 :    READ SPAN$(N)
410 :    SPAN$(N)=C3$+SPAN$(N)+C3$
420 :  NEXT N

425 ' *** Equalize length ***

430 :  FOR N=1 TO NUMBER.WORDS
440 :    E$(N)=E3$(N)+STRING$(A(N), 32)
450 :  NEXT N
460 DATA 0,2,0,2,0,2,1,1,1,1,0,3,2,2,1,3,1,0,1,1,1
470 DATA IF, SI, RUN, CORRA, INPUT, ENTRE, LIST, ALISTE, END,
    FIN, PRINT, IMPRIMA, READ, LLEVE, DATA, DATOS, THEN,
    LUEGO, FOR, PARA, STOP, CESE, NEXT, PROXIMO
480 DATA CLS, BORRE, GOTO, VAYA A, RESTORE, RESTAURE
490 DATA GOSUB, VAYA SUB, RETURN, RETORNE, ON, EN
500 DATA STEP, GRADA, REM, NOTA, LET, HACE
510 FLAG=1

515 ' *** Instructions ***

520 PRINT TAB(T)"SPANISH-ENGLISH PROGRAM TRANSLATOR"
530 PRINT TAB(T)"Do you want instructions (Y/N)?"
540 A1$=INKEY$
550 IF A1$="" THEN 540
560 IF A1$="Y" OR A1$="y" THEN 590
570 IF A1$="N" OR A1$="n" THEN CLS: GOTO 1330
580 GOTO 540
590 CLS:PRINT
600 PRINT TAB(8)"This program allows Spanish-speaking students to "
610 PRINT TAB(8)"write programs using Spanish keywords instead of"
620 PRINT TAB(8)"the English equivalents.  Most Tiny BASIC key-"
630 PRINT TAB(8)"words may be used."
640 PRINT TAB(8)"The program prepares two versions of the program"
650 PRINT TAB(8)"--  one in Spanish, and a 'translated ', English"
660 PRINT TAB(8)"version."
670 PRINT TAB(8)"Although programs may be written in Spanish, they"
680 PRINT TAB(8)"may not be RUN in that form (this is not an inter-"
690 PRINT TAB(8)"preter) until they have been translated  into Eng-"
700 PRINT TAB(8)"lish-BASIC."
710 PRINT
720 PRINT TAB(8)"Both the Spanish and  English  versions may be"
730 PRINT TAB(8)"saved to disk under filenames of your choice.  The"
740 PRINT TAB(8)"English version can then be loaded and RUN norm-"
```

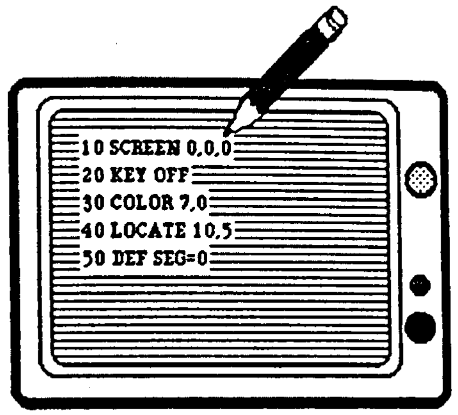
```
750 PRINT TAB(8)"ally."
760 PRINT TAB(8)"To use, type in program, using the Spanish keywords"
770 PRINT TAB(8)"where needed.  Only one statement is allowed per   "
780 PRINT TAB(8)"line.  User ";
790 COLOR 17,0
800 PRINT "must";:COLOR 7,0:PRINT" add a space after line numbers"
810 PRINT TAB(8)"and ";:COLOR 17,0:PRINT"all";:COLOR 7,0:PRINT" keywords -
even before quotation marks."
820 PRINT TAB(8)"Only line numbers between 1 and 200 may be used."
830 GOSUB 110
840 PRINT TAB(8)"To edit any line, just re-enter that line number  "
850 PRINT TAB(8)"and the new line."
860 PRINT
870 PRINT TAB(8)"Other BASICA keywords not translated may be    "
880 PRINT TAB(8)"incorporated into the program if they adhere to    "
890 PRINT TAB(8)"correct syntax.  These include :"
900 PRINT TAB(8)"ELSE,INSTR,RIGHT$,LEFT$, as well as functions,"
910 PRINT TAB(8)"(INT,RND), operators (AND,OR)."
920 PRINT  STRING$(50, 32);
930 PRINT TAB(8)"If you have any questions type either 'HELP' or"
940 PRINT TAB(8)"'AYUDA'.  You will be shown a list like these:"
950 GOSUB 110
960 GOSUB 1120
970 PRINT TAB(8)"A typical program might look something like this : "
980 PRINT  STRING$(50, 32);
990 PRINT TAB(14)"10 ENTRE ";C1$;"COMO SE LLAMA";C1$;";A$"
1000 PRINT TAB(14)"20 SI A$=";C1$;"JOSE";C1$;" VAYA A 40"
1010 PRINT TAB(14)"30 CESE"
1020 PRINT TAB(14)"40 IMPRIMA ";C1$;"HOLA JOSE";C1$"
1030 PRINT TAB(14)"50 FIN"
1040 GOSUB 110
1050 IF INKEY$ ="" THEN 1050
1060 FLAG=0
1070 CLS
1080 GOTO 1330
1090 GOSUB 1110
1100 GOTO 1330
1110 CLS
1120 PRINT " Los Mandados:"
1130 PRINT
1140 PRINT "Ahorre  (ahorrar una programma al disk)"
1150 PRINT "CARGE   (cargar una programma de disk)"
1160 PRINT "ALISTE  (Alistar una programma en espanol)"
1170 PRINT "LIST    (Alistar una programma en ingles)"
1180 PRINT "AYUDA,CORRE,NUEVO,BORRE"
1190 PRINT
1200 PRINT "Las declaraciones:"
1210 PRINT
1220 PRINT "IF=SI            RUN=CORRA         INPUT=ENTRE"
1230 PRINT "END=FIN          LIST=ALISTE       PRINT=IMPRIMA"
1240 PRINT "READ=LLEVE       THEN=LUEGO        NEXT=PROXIMO"
1250 PRINT "DATA=DATOS       GOTO=VAYA A       RESTORE=RESTAURE"
1260 PRINT "FOR=PARA         STOP=CESE         CLS=BORRE"
1270 PRINT "ON=EN            STEP=GRADA        GOSUB=VAYA SUB"
1280 PRINT "REM=NOTA         LET=HACE          RETURN=RETORNE"
1290 GOSUB 110
1300 IF FLAG=1 THEN RETURN
```

```
1310 CLS
1320 RETURN

1325 ' *** Get Keyboard Input ***

1330 PRINT">";
1340 P1=0
1350 LINE INPUT A$
1360 TEMP$=""
1370 FOR N1=1 TO LEN(A$)
1380 T$=MID$(A$,N1,1)
1390 T=ASC(T$)
1400 IF T>96 AND T<123 THEN T=T-32
1410 TEMP$=TEMP$+CHR$(T)
1420 NEXT N1
1430 A$=TEMP$
1440 COMMAND.LINE$=LEFT$(A$, 4)

1445 ' *** Check for Command ***

1450 IF COMMAND.LINE$="ALIS" THEN 1880
1460 IF COMMAND.LINE$="AHOR" THEN 2040
1470 IF COMMAND.LINE$="CARG" THEN 2170
1480 IF COMMAND.LINE$="LIST" THEN 2300
1490 IF COMMAND.LINE$="AYUD" THEN GOSUB 1110: GOTO 1330
1500 IF COMMAND.LINE$="HELP" THEN GOSUB 1110: GOTO 1330
1510 IF COMMAND.LINE$="CORR" THEN 280
1520 IF COMMAND.LINE$="NUEV" THEN 280
1530 IF COMMAND.LINE$="BORR" THEN CLS: GOTO 1330
1540 IG$=A$
1550 A$=A$+CHR$(32)
1560 B=INSTR(A$, C1$)
1570 C=INSTR(A$, C2$)
1580 IF C=0 AND B=0 THEN 1660
1590 IF B=0 THEN 1650
1600 W$=MID$(A$, B+1)
1610 P1=INSTR(W$, C1$)+B
1620 IF C<B THEN 1650
1630 IF C>P1 THEN 1650
1640 GOTO 1660
1650 IF C<>0 THEN PRINT"SOLAMENTE UNA DECLARACION CADA LINEA": GOTO 1330
1660 T$=""

1665 ' *** Check for line number ***

1670 :  FOR T=1 TO LEN(A$)
1680 :    IF MID$(A$, T, 1)=CHR$(32)THEN 1710
1690 :    T$=T$+MID$(A$, T, 1)
1700 :  NEXT T
1710 LI=VAL(T$)
1720 IF LI>L2 THEN PRINT"COMENCE LA LINEA CON UN NUMERO MENOS QUE ";L2: GOTO
1330
1730 IF LI<1 THEN PRINT"COMENCE LA LINEA CON UN NUMERO": GOTO 1330

1735 ' *** Look for Spanish keywords ***

1740 :  FOR G=1 TO NUMBER.WORDS
```

```
1750 :    P=INSTR(A$, SPAN$(G))
1760 :    IF P>0 THEN 1820
1770 :  NEXT G
1780 E2$(LI)=IG$
1790 CP$(LI)=A$
1800 IF LI>HIGH.NUMBER THEN HIGH.NUMBER=LI
1810 GOTO 1330
1820 IF P<B THEN 1850
1830 IF P>P1 THEN 1850
1840 GOTO 1770
1850 L=LEN(E$(G))

1855 ' *** Make Substitution ***

1860 MID$(A$, P, L)=E$(G)
1870 GOTO 1770

1875 ' *** List Spanish Program Lines ***

1880 V=INSTR(A$, C3$)
1890 IF V=0 THEN 1950
1900 V2$=MID$(A$, V)
1910 V3=VAL(V2$)
1920 IF V3>0 THEN PRINT E2$(V3) ELSE 1950
1930 PRINT
1940 GOTO 1330
1950 CU=1
1960 CLS
1970 :  FOR N=1 TO HIGH.NUMBER
1980 :    IF E2$(N)="" OR E2$(N)="," THEN 2020
1990 :    PRINT E2$(N)
2000 :    CU=CU+1
2010 :    IF CU/14=INT(CU/14)THEN PRINT"EMPUJE < ENTER >";: INPUT E$
2020 :  NEXT N
2030 GOTO 1330

2035 ' *** Save Programs to Disk ***

2040 INPUT"NOMBRE DE LA PROGRAMA EN ESPANOL :";NE$
2050 INPUT"NOMBRE DE LA PROGRAMA EN INGLES :";NI$
2060 OPEN"O",1, NE$
2070 :  FOR N=1 TO 200
2080 :    PRINT#1, E2$(N); CHR$(13);
2090 :  NEXT N
2100 CLOSE 1
2110 OPEN"O",1, NI$
2120 :  FOR N=1 TO 200
2130 :    PRINT#1, CP$(N); CHR$(13);
2140 :  NEXT N
2150 CLOSE 1
2160 GOTO 1330

2165 ' *** Load Programs From Disk ***

2170 INPUT"NOMBRE DE LA PROGRAMA EN  ESPANOL :";F$
2180 INPUT"NOMBRE DE LA  PROGRAMA  EN INGLES :";F3$
2190 OPEN"I",1, F$
```

```
2200 :  FOR N=1 TO 200
2210 :    LINE INPUT #1, E2$(N)
2220 :  NEXT N
2230 CLOSE 1
2240 OPEN"I",1, F3$
2250 :  FOR N=1 TO 200
2260 :    LINE INPUT #1, CP$(N)
2270 :  NEXT N
2280 CLOSE 1
2290 GOTO 1330
2300 CU=1

2305 ' *** List Programs ***

2310 :  FOR N=1 TO HIGH.NUMBER
2320 :    IF CP$(N)<>"" THEN PRINT CP$(N): CU=CU+1
2330 :    IF CU/14=INT(CU/14)THEN PRINT"EMPUJE < ENTER >";: INPUT E$
2340 :  NEXT N
2350 PRINT
2360 GOTO 1330
2370 CLOSE:END
```

Chapter 13



Indexer

Are you using your IBM PC or PCjr to write a term paper, article, or book? If so, you may need to prepare an index or glossary for your project. Or, are you curious about the scope of your vocabulary? These two programs, Index 1 and Index 2, will make your work quite a bit easier. Index 1 will take most reasonably-sized documents—text or program files both—and throw out the punctuation marks and numbers. DOS will SORT this list for you, and then Index 2 will go through it and discard duplicates and many plurals of a root word. You wind up with an alphabetized listing only of the unique words in your document.

I recently wrote these programs to help in the preparation of a book I was working on. I ran about 60,000 words through them and ended up with a list of a few thousand unique words that I further condensed to form my glossary and index. The Indexer programs will also work with your shorter text items, such as letters, short stories, or school assignments. Odd punctuation won't throw it, and capitalized words are automatically converted to lowercase. You can even use the program on your BASIC programs to find out what keywords were used. Line numbers and other nonalpha characters will be discarded, as well.

INDEX 1: PREPARING THE FILE FOR SORTING

Indexer might be a candidate for the misnomer of the year award. But then, Lotus 1-2-3 doesn't have anything to do with yoga, either. The BASIC programs themselves don't really index or sort anything, although that is the end result. Instead, they serve as a preliminary "filter" to strip off unwanted characters and numbers, in effect deciding what is a word and what isn't. The legal words are written to disk, where DOS's SORT filter can rearrange the list in alphabetical order.

Because of the size of the files processed by this program, I didn't bother with including a BASIC sort routine,which would be much too slow. The program could take hours to sort such a huge list in memory using only Basic techniques. Since In-

```
a                can              file             letting
about            candidate        filename         line
accomplished     capitalized      files            list
added            chapter          filter           listing
all              character        find             lot
allow            characters       first            lowercase
alone            check            following        machine
alphabetical     chr              follows          made
alphabetized     code             for              make
also             comes            form             mandatory
although         command          from             many
america          compare          furnished        mark
amplitude        computers        further          marks
an               condensed        glossary         may
and              conversion       go               million
another          converted        goes             minutes
any              curious          gotten           misnomer
anything         deciding         greater          most
anyway           different        had              much
appeared         discard          has              my
are              discarded        have             mydoc
array            disk             help             mylist
article          do               i                need
as               document         ibm              new
ascii            documents        if               newly
assignments      does             in               next
at               dos              including        non
automatically    down             index            note
award            drive            indexer          nothing
awhile           duplicate        individual       numbers
back             duplicates       initial          odd
basic            each             instead          of
be               easier           instr            off
because          effect           into             on
been             else             is               once
beginning        end              it               one
between          ended            items            only
bit              ends             keywords         ooops
book             enter            language         or
both             entering         last             order
bother           equivalent       learn            other
broken           even             left             our
built            fast             legal            out
by               few              less             paper
called           figured          letters          parsing
```

Fig. 13-1. The sorted list of words in this chapter.

```
pc               result           taken            very
pcjr             root             takes            view
perform          routine          temporarily      vocabulary
place            run              term             wanted
plurals          school           text             was
position         scope            than             we
preliminary      search           that             well
preparation      see              the              were
prepare          separate         them             what
process          serve            themselves       when
processed        sets             then             where
produced         short            these            which
producing        shorter          they             will
program          should           this             willing
programs         since            thousand         wind
project          size             through          with
properly         slow             throw            won
punctuation      so               to               word
quite            something        too              wordlist
ran              sort             two              words
read             sorted           type             work
reads            space            under            working
really           spaces           unique           would
rearrange        stored           unwanted         wrd
reasonably       stories          up               write
recently         string           uppercase        written
recommend        strip            us               wrote
redefined        such             use              year
remainder        symbol           used             yet
remove           syntax           users            you
requires         system           using            your
reside           take             variable
```

dexer requires a disk drive anyway, I figured most users would be willing to learn to use DOS's built-in sort. The DOS sort routine is fast and efficient. It requires, however, that the indexing process take place in two parts. First the file is prepared for sorting, DOS sorts it, and then the sorted file is examined for unique words.

Although the machine language sort of your word array by DOS is very fast, parsing the document into individual words takes a few minutes. I'd recommend letting the program run for awhile as you do something else.

The program Index 1 reads in each line of your file, in line 260, and sets the initial search position for spaces, P, at one. Then INSTR is used to find the first occurrences of a space, which is used by this program to mark the ends of words. A check is made to remove any punctuation marks that may have been "attached" to the ends of our words.

The newly found word is stored temporarily in variable WRD$, and A$ is redefined as the remainder of the string following WRD$. WRD$ is then converted to all lowercase letters, to allow us to sort words like "America," "DOS," and "amplitude" properly. Computers see uppercase and lowercase letters as different, so that a follows Z. The conversion is accomplished in a routine beginning at line 350, where the ASCII code for each character is figured. If it is greater than CHR$(64) and less than CHR$(92) then 32 is added to the character, producing the lowercase equivalent. All other characters are left alone.

The newly lowercased WRD$ is written to disk in line 450, and the program goes back to process the next word.

When your file has been broken down into separate words by Index 1, you can then go to DOS, using the SYSTEM command, and perform the SORT.

THE DOS SORT

SORT.EXE, furnished with DOS, should reside on your disk. You enter the file name your word list is stored under and then a new file name for the sorted word list, using the following syntax:

```
SORT <filetosort >sortedfile
```

Note that the space between the file name-to-sort and the greater than symbol (>) is mandatory. If you had run Index 1 on MYDOC.TXT and produced MYLIST.TXT, and wanted the sorted result to go to a disk file called RESULT, you would enter:

```
SORT <MYLIST.TXT >RESULT
```

INDEX 2: PRODUCING THE FINAL LIST

Once the sort has taken place, you can view the list by entering TYPE RESULT from DOS. Oops. A lot of duplicate words in the file? A million A's? A thousand occurrences of the? That's where Index 2 comes in. This program does nothing more than read in your sorted RESULT wordlist and compare each word with the last one. Only if they are different will the program write the word out to yet another disk file. When this is accomplished, you will have a sorted, alphabetized list of unique words that appeared in your document.

A sorted list of the words in this chapter, processed by Indexer is shown in Fig. 13-1.

Listing 13a: The Index 1 Program

```
10 ' ********************
20 ' *                  *
30 ' * Indexer Part One *
40 ' *                  *
50 ' ********************

55 ' *** Initialize ***

60 SCREEN 0,0,0
70 COLOR 7,0
80 KEY OFF
90 ON KEY(10) GOSUB 570
100 KEY(10) ON
110 CLS
120 LOCATE 25,30
130 COLOR 16,7
140 PRINT" Hit F10 to abort. ";
```

```
145 ' *** Enter filenames ***

150 COLOR 7,0
160 LOCATE 2,1
170 CU=0
180 INPUT"ENTER FILENAME TO PROCESS :";F$
190 INPUT"ENTER OUTPUT FILENAME";F1$
200 OPEN "I",1,F$
210 OPEN "O",2,F1$
220 LOCATE 25,10
230 COLOR 16,7
240 PRINT " READING/WRITING FILE : ";
250 COLOR 7,0

255 ' *** Read in a line ***

260 LINE INPUT#1,A$
270 P=1
280 R=INSTR(P,A$,CHR$(32))
290 IF R=0 THEN GOTO 480
300 WRD$=LEFT$(A$,R-1)
310 A$=MID$(A$,R+1,255)
320 IF WRD$="" THEN GOTO 270
330 WRD$=T2$+WRD$
340 T2$=""

345 ' *** Change to lowercase ***

350 FOR N=1 TO LEN(WRD$)
360 T$=MID$(WRD$,N,1)
370 T=ASC(T$)
380 IF T>64 AND T<92 THEN T=T+32
390 IF T<97 OR T>122 THEN GOTO 420
400 TEMP$=TEMP$+CHR$(T)
410 NEXT N
420 WRD$=TEMP$
430 LOCATE 25,10
440 TEMP$=""

445 ' *** Print word to disk ***

450 PRINT #2,WRD$
460 CU=CU+1
470 GOTO 270
480 T2$=A$
490 IF EOF(1) GOTO 510
500 GOTO 260
```

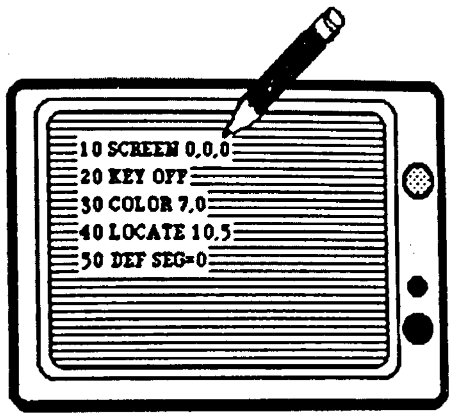
```
510 CLOSE

515 ' *** Show results ***

520 LOCATE 25,10
530 PRINT SPACE$(40);
540 LOCATE 25,10
550 PRINT"FINISHED    -- ";
560 PRINT CU;" words found.";
570 CLOSE
580 END
```

Listing 13b: The Index 2 Program

```
10 ' ********************
20 ' *                  *
30 ' * Indexer Part Two *
40 ' *                  *
50 ' ********************

55 ' *** Initialize ***

60 SCREEN 0,0,0
70 COLOR 7,0
80 KEY OFF
90 ON KEY(10) GOSUB 390
100 KEY(10) ON
110 CLS
120 LOCATE 25,30
130 COLOR 16,7
140 PRINT" Hit F10 to abort. ";

145 ' *** Enter filenames ***

150 COLOR 7,0
160 LOCATE 4,1
170 CU=0
180 INPUT"ENTER FILENAME TO PROCESS :";F$
190 INPUT"ENTER OUTPUT FILENAME";F1$
200 OPEN "I",1,F$
210 OPEN "O",2,F1$
220 LOCATE 25,10
230 COLOR 16,7
240 PRINT " READING/WRITING FILE : ";
250 COLOR 7,0
```

```
255 ' *** Read a word ***

260 LINE INPUT#1,A$
270 IF A$<>LAST$ GOTO 290
280 GOTO 260

285 ' *** Write Unique Word ***

290 PRINT #2,A$
300 CU=CU+1
310 LAST$=A$
320 IF EOF(1) GOTO 340
330 GOTO 260

335 ' *** Show Results ***

340 LOCATE 25,10
350 PRINT SPACE$(40);
360 LOCATE 25,10
370 PRINT"FINISHED --- ";
380 PRINT CU;" unique words found."
390 CLOSE
400 END
```

Chapter 14



Error Trapper

Error Trapper is dedicated to all of you who have written programs containing a bug or two. Readers who have never made a mistake in their programs can skip this chapter and go on to the next. Okay, who's left?

I'll address the rest of this chapter to the three or four readers who occasionally make a mistake in their programming. Error Trapper is especially aimed at those of you who are very creative in their errors, and who trigger some the more obscure error messages, like:

```
UNIX COMMAND: NOT RECOGNIZED.
PLEASE RELACE SHELL
```

Don't go scrambling for your manuals. I just made that up. But do you understand all of the error messages that you DO see? This is quite a long program—more than 300 lines—but you only have to type in Error Trapper once. Then you can use it to teach new programmers or to avoid having to reach for the BASIC manual every time an error occurs during program writing.

Our handy Basic interpreters are nice enough to point errors out to us during runtime. It would have been nice to have the syntax errors, at least, brought to our attention when the program line was first entered. But, no, the computer is not that courteous. (Some computers actually do this, though.) Instead, the IBM PC reserves judgement until we actually try to run the program.

Few amateur programs and darned few professional BASIC programs take advantage of the error trapping possibilities of the IBM PC. I haven't used ON ERROR much in this book. Instead, I have tried to anticipate what errors might be made and prevent them where possible. Many programs won't accept improper input, or prompt you for the type of information you should enter next. This concept has been carried to extremes in Music Writer, in Chapter 19.

Sometimes an unanticipated error takes place. Usually this will be in a poorly debugged program that does not have sufficient error traps built in. Often the errors will occur during program debugging itself.

Ordinarily when an error takes place, the computer will stop the program at that point and deliver a one-line message outlining the error, such as "NEXT without FOR." You can, however, use the ON ERROR interrupt to send the program on to a special error trapping routine. If, say, the error is "File not Found," your program can display a message like "Insert the Automatic Programmer Disk in Drive B, and press ENTER." That way, the user is not dumped out of the program into BASIC without a hint of what to do next.

The PC is nice enough not only to tell us that there is an error, but also to point out exactly what type of error has been made. A clever error code number, which can be manipulated by the program, is supplied. In many cases, a routine, like the one described above, could be written to recover from the error. Or, in other cases, the error number could be used to supply the operator with some hint of what he or she has done wrong.

For another example, you could supply a friendly prompt on the order of "Program tried to divide by zero. Are you sure all the amounts you entered are correct?" would be nice. Admittedly, many programmer's don't understand enough about errors to do anything about them.

ERROR TRAPPER MESSAGES

That's where Error Trapper comes in. BASIC does provide nice long error messages. Some of the more esoteric error messages, however, may puzzle the best of us. Do you really know what sort of mistake will trigger an "Illegal Direct" message?

This program, when appended to your own program, will spell it out for you. It provides REALLY long error messages. Instead of just telling you how you goofed, it will suggest situations that might have produced the error and places to check for the bug.

For example, if you see "Out of Data," you know, in fact, that the computer would like more data items. Error Trapper suggests that perhaps several data items were left out by mistake, or that the FOR-NEXT loop that reads the data is too large.

"Illegal Function Call" suggests that the programmer list the offending line, and print out from command mode some of the values of the variables.

THE INTERRUPT ROUTINE

Little understood is how the IBM PC manages to do something about errors. The secret is in line 10070, which is an ON ERROR GOTO . . . command that summons the computer's interrupt routine. Interrupts, as you have seen, are different than normal statements. If a program line says IF INKEY$ = " " GOTO, it will act on that only at the exact moment that the line is interpreted by Basic. In order to make INKEY$ work, the program has to loop back, over and over, until something happens.

Once ON ERROR has been activated, however, the computer can go on to the other things. The program can perform all sorts of different functions, and the interrupt routine will remain dormant... until an error occurs. Then it will obey the command and send control to the line previously specified.

You can't even turn the interrupt routine off by exiting the program. Try this: run Error Trapper, and press break at some point. Then, trigger an error by typing in a syntax error or some other goof from command mode. Oops! The program is running again, and you are at line 10080. You didn't even type RUN. That is the interrupt routine at work.

Once an error has taken place, Error Trapper looks to see what kind of error it is. An error deposits a value in the reserved variable ERR. The same error always produces the same unique number. So, I use that number in this program in an ON . . GOSUB line that directs control to the appropriate error message. In a real program, you might substitute some type of error trap for the message. The trap might be a routine that corrects the error.

For example, if the error were "File Not Found," you might write a routine that asks the user to check the file name or deposit the correct disk in the drive. Then it would ask again for the file name. Using RESUME, followed by a line number, control can be returned to the main body of the program.

If you append Error Trapper to your own programs, you will want to move the early parts of this program, such as the ON ERROR line, and the DIMension statement, as well as the READ loop, earlier in the program, so they will be activated BEFORE the main body of the program is run.

Listing 14: The Error Trapper Program

```
10000 '     ****************************
10010 '     *                          *
10020 '     *      Error Trapper       *
10030 '     *                          *
10040 '     ****************************
10050 '
10060 DIM ER(51)
10070 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
10080 DATA 20,22,23,24,25,26,27,29,30,50,51,52,53,54,55,57,58
10090 DATA 61,62,63,64,66,67,68,69,70,71,72,73,74,75,76
10100 FOR N=1 TO 51
10110 READ ER(N)
10120 NEXT N
10130 CLS:PRINT
10140 ON ERROR GOTO 10160
10150 GOTO 10150 ' *** BRANCH TO YOUR PROGRAM ***
10160 FOR N=1 TO 51
10170 IF ER(N)=ERR THEN EC=N:GOTO 10210
10180 NEXT N
10190 PRINT "Unprintable error"
10200 RESUME
10210 CLS
10220 ON EC GOSUB 10240,10300,10340,10400,10460,10560,10640
      ,10710,10800,10900,11000,11070,11100,11160,11240
      ,11310,11370,11420,11480,11510,11550,11610,11670,
       11740,11760,11810,11860,11900,11920,11950
10230 ON EC-30 GOSUB 11980,12050,12090,12160,12220,12250,
      12310,12340,12420,12450,12520,12590,12630,
      12660,12780,12810,12840,12890,12920,12960,12990
10240 PRINT "Next without For"
10250 PRINT:PRINT"Program got to NEXT without encountering FOR first"
10260 PRINT"Check for incorrect GOTO.  Also, did you type GOTO from"
10270 PRINT"COMMAND mode?"
10280 PRINT
10290 GOTO 13020
10300 PRINT"Syntax error"
10310 PRINT "Check for misspelled keywords, missing parentheses or quotes"
10320 PRINT "as well as bad punctuation."
10330 GOTO 13020
10340 PRINT "Return Without Gosub"
10350 PRINT "Program may have gotten to a subroutine improperly."
10360 PRINT "Check program lines immediately prior to this subroutine"
10370 PRINT "to make sure program control does not allow running"
10380 PRINT "into the following module."
10390 GOTO 13020
```

```
10400 PRINT "Out of Data"
10410 PRINT "Program was asked to Read more data items than were"
10420 PRINT "available.  Check Data lines to be sure that none"
10430 PRINT "were left out by mistake.  FOR-NEXT loop may also"
10440 PRINT "be too large for number of items in Data."
10450 GOTO 13020
10460 PRINT "Illegal Function Call"
10470 PRINT "Program tried to perform an operation using an illegal"
10480 PRINT "parameter.  Print the values of the variables in the"
10490 PRINT "program line.  One will probably be a value that is "
10500 PRINT "unsuited for one of the functions of that line."
10510 PRINT "For example, you might have PEEK(N) in the line, and"
10520 PRINT "discover than N equals 70,000.   Or, in the case of"
10530 PRINT "PRINT CHR$(N) that, through some error in the program,"
10540 PRINT "N equals 256, or a larger number."
10550 GOTO 13020
10560 PRINT "Overflow"
10570 PRINT "A number is too large. If a variable is an integer,"
10580 PRINT "this will occur if the number is larger than 32767"
10590 PRINT "single or double precision numbers can only be in the range"
10600 PRINT "of about 1.7E+ (or minus) 38.  By changing a variable "
10610 PRINT "from integer to single or double precision, most "
10620 PRINT "overflow errors will be avoided."
10630 GOTO 13020
10640 PRINT "Out of Memory"
10650 PRINT "Most likely, your program uses up too much memory "
10660 PRINT "because of very large arrays.  Cut down on array size"
10670 PRINT "if possible.  Improperly nested branching routines"
10680 PRINT "(10 GOSUB 10, in the worst possible case) can also"
10690 PRINT "cause this, but rarely."
10700 GOTO 13020
10710 PRINT "Undefined line"
10720 PRINT "You typed a GOTO or GOSUB line, without entering "
10730 PRINT"the line where control was directed.  Or, in editing,"
10740 PRINT "you killed a program section without the corresponding"
10750 PRINT "line which called that section.   It is a good idea"
10760 PRINT "to use a cross-reference utility to find out if a"
10770 PRINT" program line is called from elsewhere in a program"
10780 PRINT "before killing it."
10790 GOTO 13020
10800 PRINT "Subscript Out of Range"
10810 PRINT "Program tried to use an array element larger than was"
10820 PRINT "DIMensioned.  Print out current value of the subscript"
10830 PRINT "in the affected program line.  If it is 11, you may"
10840 PRINT "have forgotten to DIMension that array, or you have"
10850 PRINT "spelled the array name differently in the program line."
10860 PRINT "For example:"
10870 PRINT "10 DIM ST$(20)"
10880 PRINT "20 S2$(12)=A$"
10890 GOTO 13020
10900 PRINT"Duplicate Definition"
10910 PRINT "Redimensioned Array"
10920 PRINT "Place DIM statements at beginning of program, where"
10930 PRINT "they are not likely to be encountered more than once."
10940 PRINT "If a program will be repeated, use the RUN command"
10950 PRINT"or make sure the GOTO directs control AFTER the DIM statement."
10960 PRINT "If an array is being DIMensioned with a variable,"
```

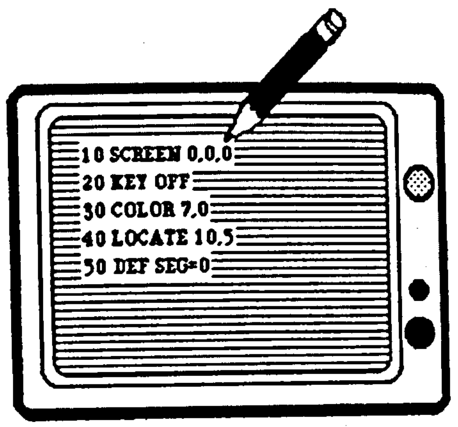
```
10970 PRINT "(as in DIM A$(N)), make sure that the variable has been"
10980 PRINT "assigned a value earlier in the program"
10990 GOTO 13020
11000 PRINT"Division by Zero"
11010 PRINT "Program error has produced a zero value in a variable"
11020 PRINT"that is used in a division operation.  Check variable"
11030 PRINT"to make sure it is not spelled incorrectly or that the"
11040 PRINT"wrong variable is not being used.  Find  out why it is"
11050 PRINT"zero when a value was expected."
11060 GOTO 13020
11070 PRINT"Illegal direct"
11080 PRINT "The INPUT command cannot be used as a direct command."
11090 GOTO 13020
11100 PRINT"Type Mismatch"
11110 PRINT "Program tried to assign a string value to a numeric"
11120 PRINT "variable or vice versa.  For example:  A$=A, or A=CHR$(N)."
11130 PRINT "In most cases, these are caused by forgetting to include"
11140 PRINT "the $ in a string variable or array."
11150 GOTO 13020
11160 PRINT "Out of String Space"
11170 PRINT"Unlike some other BASICs, IBM BASIC allocates the"
11180 PRINT"Needed memory for strings dynamically.  If you see"
11190 PRINT"this message, then your string variables caused BASIC"
11200 PRINT"to exceed the amount of memory left, even after string"
11210 PRINT"garbage collection.  Look for ways to reduce the size"
11220 PRINT"of your program."
11230 GOTO 13020
11240 PRINT"String Too Long"
11250 PRINT"String variables and array elements can only be 255 bytes"
11260 PRINT "long.  Take string variables in program line, and "
11270 PRINT "find length by typing PRINT LEN(variable$) in command"
11280 PRINT "mode.  The find why attempt was made to make this"
11290 PRINT "string that long."
11300 GOTO 13020
11310 PRINT "String Formula Too Complex"
11320 PRINT "Avoid such complex formulae as:"
11330 PRINT " A$=(LEFT$(MID$(A$,INSTR(B$,C$),LEN(A$))-1,) LEN(A$)
11340 PRINT "Break operations down into several components.  You will"
11350 PRINT "Never get all the parentheses in the right places, anyway."
11360 GOTO 13020
11370 PRINT "Can't Continue"
11380 PRINT "Either you typed CONT after program had ended, or "
11390 PRINT "a program line was edited (thus ending the program)."
11400 PRINT "You must start the RUN over."
11410 GOTO 13020
11420 PRINT"Undefined User Function"
11430 PRINT"You tried to call a DEF FN function that has not"
11440 PRINT"been defined.  Check your spelling to make sure that"
11450 PRINT"the function you have defined is spelled the same as"
11460 PRINT"the one you are calling."
11470 GOTO 13020
11480 PRINT "No Resume"
11490 PRINT "Program ended during error trapping."
11500 GOTO 13020
11510 PRINT"Resume Without Error"
11520 PRINT "You forgot, deleted, or bypassed the necessary "
11530 PRINT "ON ERROR GOTO message.  Place early in program."
```

```
11540 GOTO 13020
11550 PRINT"Missing Operand"
11560 PRINT "Program neglected to include one of the necessary operands."
11570 PRINT "Examples:"
11580 PRINT "A$=LEFT$(A$)"
11590 PRINT "POKE 28513 "
11600 GOTO 13020
11610 PRINT"Line buffer overflow"
11620 PRINT"A line can only be 255 characters.  You tried"
11630 PRINT"to enter one longer than that.  Try using
11640 PRINT"multiple statements on different lines.  Reduce"
11650 PRINT"the size of your variable names."
11660 GOTO 13020
11670 PRINT"Device timeout"
11680 PRINT"BASIC will only wait for input from a device, such"
11690 PRINT"as the Asynchronous Adapter for a fixed period of"
11700 PRINT"time.  If no input is received in that span, you"
11710 PRINT"will receive this error message.  If using COM,"
11720 PRINT"check your cables.  Try again."
11730 GOTO 13020
11740 PRINT"Device Fault"
11750 GOTO 11680
11760 PRINT"FOR without NEXT"
11770 PRINT"You will only see this error message if a"
11780 PRINT"program ends with BASIC in the middle of a"
11790 PRINT"FOR-NEXT loop.  This may or may not be a problem."
11800 GOTO 13020
11810 PRINT"Out of paper"
11820 PRINT"Your printer has sent a signal to your computer indicating it is"
11830 PRINT"out of paper.  If you receive this message and there"
11840 PRINT"is paper in your printer, check for hardware fault."
11850 GOTO 13020
11860 PRINT"WHILE without WEND"
11870 PRINT"Check for improper branching within your program"
11880 PRINT"loop improperly."
11890 GOTO 13020
11900 PRINT"WEND without WHILE"
11910 GOTO 11870
11920 PRINT"Field Overflow"
11930 PRINT "More than 255 bytes were allocated to a random-access buffer."
11940 GOTO 13020
11950 PRINT"Internal error"
11960 PRINT "Whoops.  Disk operating system goofed."
11970 GOTO 13020
11980 PRINT "Bad File Number"
11990 PRINT "File buffer number that has not been assigned with an"
12000 PRINT "OPEN statement was used.  Example:"
12010 PRINT "10 OPEN ";CHR$(34);"O";CHR$(34);",1,F$"
12020 PRINT "20 PRINT #2,A$"
12030 PRINT "Note that PRINT #1 should have been used instead."
12040 GOTO 13020
12050 PRINT"File Not Found"
12060 PRINT "File by that name not on disks currently in drive(s)."
12070 PRINT "Or, you spelled filename wrong."
12080 GOTO 13020
12090 PRINT"Bad File Mode"
12100 PRINT "You tried to write to a buffer that had been opened for"
```

```
12110 PRINT "input, or vice versa. Example:"
12120 PRINT "10 OPEN ";CHR$(34);"O";CHR$(34);",1,F$"
12130 PRINT "20 PRINT #1,A$"
12140 PRINT "Change the O to I"
12150 GOTO 13020
12160 PRINT"File already open"
12170 PRINT"An OPEN statement was encountered for a sequential"
12180 PRINT"file that was already open.  Or, you tried to kill an"
12190 PRINT"open file.  Look for improper GOTOs or GOSUBs that would"
12200 PRINT"send the program back to the OPEN statement."
12210 GOTO 13020
12220 PRINT"Disk I/O error"
12230 PRINT "OOOPS! Another computer error."
12240 GOTO 13020
12250 PRINT"File already exists."
12260 PRINT"You will only see this message if you are using the"
12270 PRINT"NAME command, and the name you specify matches a filename"
12280 PRINT"already being used on the disk.  KILL the old file if you"
12290 PRINT"do not want it, and try again.  Or use a different filename."
12300 GOTO 13020
12310 PRINT"Disk Full"
12320 PRINT "Insert new disk, or kill files."
12330 GOTO 13020
12340 PRINT"Input Past End"
12350 PRINT "Program tried to load more data from disk than was"
12360 PRINT "Available.  Check for empty file, or FOR-NEXT loop that"
12370 PRINT "is too large.  With sequential files that grow, adding"
12380 PRINT " an IF EOF(file buffer) GOTO xxx statement can check"
12390 PRINT "for the end of the file, and send control to the next"
12400 PRINT "module in an orderly manner."
12410 GOTO 13020
12420 PRINT"Bad Record Number"
12430 PRINT "Record number in a PUT statement larger than 1,340"
12440 GOTO 13020
12450 PRINT"Bad Filename"
12460 PRINT "Filename not legal.  Must conform to all rules for naming"
12470 PRINT "programs or files in Disk Basic. "
12480 PRINT "If variable being used for file name"
12490 PRINT "check to make sure illegal value not being assigned."
12500 PRINT "Error traps can be made to check for name legality."
12510 GOTO 13020
12520 PRINT"Direct Statement In File"
12530 PRINT "You cannot load, run, or merge a disk file that is not"
12540 PRINT "a Basic program.   This occurs when attempting to load"
12550 PRINT "a text file stored in ASCII form or, possibly an actual"
12560 PRINT "program that has had a line number removed from the"
12570 PRINT "beginning of the line."
12580 GOTO 13020
12590 PRINT"Too Many Files"
12600 PRINT "There is no more directory space on your diskette, or"
12610 PRINT"your file name is invalid.  Try new disk or name."
12620 GOTO 13020
12630 PRINT"Device unavailable."
12640 PRINT"Check to see if disk is in drive, door open, printer on."
12650 GOTO 13020
12660 PRINT"Communication buffer overflow."
12670 PRINT"Your program has not read all the data already in the"
```

```
12680 PRINT"communications buffer before trying to load more to it."
12690 PRINT"You can RESUME at a point that will allow clearing the"
12700 PRINT"buffer before trying to input additional data."
12710 PRINT" You can also enlarge the communications buffer by using"
12720 PRINT "/C: when entering BASIC.  Your program can also arrange"
12730 PRINT "to exchange stop/start signals such as Control-S and Control-Q"
12740 PRINT" with the other computer to keep the buffer from become too full.
12750 PRINT" Or, try using a lower baud rate.  BASIC can only handle
information"
12760 PRINT"so fast."
12770 GOTO 13020
12780 PRINT"Disk Write Protected"
12790 PRINT "Write protect notch is covered.  Or disk is in upside down!"
12800 GOTO 13020
12810 PRINT"Disk not ready"
12820 PRINT"Drive door open, or no disk in drive."
12830 GOTO 13020
12840 PRINT"Disk media error."
12850 PRINT"Your diskette may be bad.  Copy any files"
12860 PRINT"you can to a new disk.  Try to reformat to"
12870 PRINT"see if disk is still usable."
12880 GOTO 13020
12890 PRINT"You tried to use an Advanced BASIC feature from"
12900 PRINT"Disk Basic.  Load proper BASIC, and reload program."
12910 GOTO 13020
12920 PRINT"Rename across disks."
12930 PRINT"You tried to rename a file, but specified the"
12940 PRINT"wrong disk.  Try again."
12950 GOTO 13020
12960 PRINT"Path/file access error."
12970 PRINT"You tried to use a path or filename to an inaccessible file."
12980 GOTO 13020
12990 PRINT"Path not found."
13000 PRINT"DOS could not find the path you used. Try again."
13010 GOTO 13020
13020 PRINT"This error occurred in line ";ERL
13030 RESUME 10140
```

Chapter 15



Visual Maker

Though photographic in nature, conventional slide shows used in business presentations rely more on text material, charts, and graphs than on actual pictorial subjects. Visual Maker is a program written for the IBM PC that allows designing a series of text and graphics "frames," specifying how long each should appear on the CRT screen, and assembling them into a finished slide show.

HOW TO DESIGN FRAMES

Absolutely no user programming is required. The operator simply "draws" on the CRT screen, using the arrow keys for cursor control, and placing alphanumeric characters and any type of graphic blocks wherever desired. Then, the F1 is hit, and that frame is stored to disk. Then a BASIC program that will display the frames as desired in a completed, ready-to-run slide show is written.

Visual Maker is similar in concept to Screen Editor, which writes BASIC subroutines that reproduce desired instructional screens. In fact, I used Screen Editor to write all the instructions in Visual Maker. The idea is to allow the user to enter various parameters and then have the computer generate BASIC code automatically.

You can use Visual Maker in several flexible ways. It can be used to generate a slide show, start to finish. Or, you can create one frame at a time to build a slide "library." Then individual frames can be renumbered appropriately and assembled into a finished show. Thus, you may design several dozen or several hundred frames that can be used and reused in multiple slide programs. A sample slide sequence program produced by Visual Maker is provided in Fig. 15-1.

Striking an alphanumeric key reproduces that symbol on the screen, much like a word processing program. In addition, any of the available graphic characters can be summoned by hitting the ESC key followed by an alphanumeric key. You get out of graphics mode by pressing ESC a second time.

The screen editor written for Visual Maker is a fairly simple one. Exiting from a given screen line

```
10  CLS
20 PRINT TAB(7)"This is a sample frame produced by Visual Maker. "
30 PRINT TAB(7)"I am writing these directions on the screen of "
40 PRINT TAB(7)"my IBM PC.  When I am finished, I will press "
50 PRINT TAB(7)"F10, and this frame will be written to disk. "
60 PRINT
70 PRINT
80 PRINT
90 PRINT
100 PRINT TAB(10)"Graphics may also be inserted into the "
110 PRINT TAB(10)"frames, although, because my daisy-wheel "
120 PRINT TAB(10)"printer cannot reproduce graphics, I will "
130 PRINT TAB(10)"not use any in this sample frame. "
140 PRINT
150 PRINT TAB(10)"---------------------------------------------- "
160 PRINT TAB(10)"+ In many cases, normal symbols can substitute  + "
170 PRINT TAB(10)"+ for graphics anyway....                       + "
180 PRINT TAB(10)"---------------------------------------------- "
190 PRINT
200 PRINT
210 PRINT
220 PRINT
230 PRINT
240 PRINT
250 F=TIMER+10
260 IF TIMER<F THEN GOTO  260
270 CLS
280 PRINT TAB(2)"This is the second frame in the sample program. "
290 PRINT
300 PRINT
310 PRINT
320 PRINT
330 PRINT
340 PRINT TAB(5)"I intend to have this displayed for 10 seconds. "
350 PRINT
360 PRINT
370 PRINT
380 PRINT
390 PRINT
400 PRINT
410 PRINT
420 PRINT
430 PRINT
440 PRINT
450 PRINT
460 PRINT
470 PRINT
480 PRINT
490 PRINT
500 PRINT
510 F=TIMER+10
520 IF TIMER<F THEN GOTO  520
530 CLS
```

Fig. 15-1. Variables used in Visual Maker.

should only be done at a point in which a space already exists, otherwise the character left in the cursor position will be erased. Or, the line can be finished. The cursor will wrap around to the next. The graphic blocks can be used to build charts, graphs, and other material. When you are satisfied with the screen design, hit F1.

DISPLAY TIME

At this point, the program uses the SCREEN function to look at each position on the screen, building a line in a manner identical to the method used in Screen Writer. The main change is that each screen program line set is concluded with a line that limits the amount of time that screen is displayed.

You will be asked how long you want the slide shown. That number, LENGTH$, is used to write a line that constructs a line that checks the value of TIMER during the display of the slide. As long as TIMER is less than F, which is its value when the slide was first displayed plus the value of LENGTH$, the frame will continue to be displayed.

TIMER, as you may know, keeps track of elapsed seconds in 0.01 second increments. You can use this reversed variable in our own programs. A typical use is to get a random number to reseed the random number generator for games programs:

```
100 RANDOMIZE TIMER
```

You can also use TIMER with an interrupt routine to send a program to a desired subroutine when a given number of seconds has elapsed.

```
100 ON TIMER(60) GOSUB 200
110 TIMER ON
...
...
200 PRINT "One minute has passed!"
```

This type of routine is valuable because our program can be doing other things while TIMER is ticking off. With Visual Maker, however, we don't want anything to happen other than the image to remain on the screen. So, the delay is something like this:

```
100 INPUT "Enter number of
    seconds to delay ";I
110 F=TIMER+D
120 PRINT"This is the image!"
130 IF TIMER<F GOTO 130
150 CLS
160 PRINT "Image displayed for
    ";D;" seconds"
```

That's roughly what's done for each frame produced by Visual Maker, except that the value for D is entered once, and the program line built from that information. When the slide show is run, each frame will be shown for the desired interval. When the specified time has elapsed, the screen is cleared, and the program goes on to the next frame.

Listing 15: The Visual Maker Program

```
10 '     ************************
20 '     *                      *
30 '     *   Visual Maker       *
40 '     *                      *
50 '     ************************
60 '

65 ' *** Initialize ***

70 DEFINT A-Y
80 ROW=1:COL=1
90 WHITE=219
```

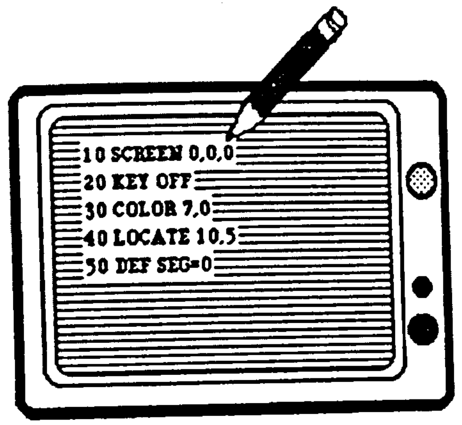
```
100 DIM LN$(400)
110 KEY OFF
120 SCREEN 0,0,0
130 COLOR 7,0
140 ON KEY(1) GOSUB 1390
150 ON KEY(10) GOSUB 2120
160 KEY(10) ON
170 ON KEY(11) GOSUB 1330
180 ON KEY(12) GOSUB 1230
190 ON KEY(13) GOSUB 1180
200 ON KEY(14) GOSUB 1280
210 WIDE=80
220 GOTO 250
230 A$=INKEY$:IF A$="" GOTO 230
240 RETURN
250 SP$=CHR$(32)
260 CLS
270 CU=1
280 :  FOR N8=1 TO 100
290 :    LN$(N8)=""
300 :  NEXT N8
310 LN=10:IC=10
320 PRINT:PRINT:PRINT
330 GOSUB 350
340 GOTO 470

345 ' *** Enter filename of program ***

350 LINE INPUT"ENTER FILE NAME  :  ";F$
360 FOR N=1 TO LEN(F$)
370 T=ASC(MID$(F$,N,1))
380 IF T>96 AND T<123 THEN MID$(F$,N,1)=CHR$(T-32)
390 NEXT N
400 IF LEN(F$)>12 THEN PRINT"File name too long!":PRINT:
     GOTO 350
410 S9=INSTR(F$,".BAS")
420 IF LEN(LEFT$(F$,S9))>8 THEN PRINT"File name too
     long!":PRINT:GOTO 350
430 IF S9=0 THEN PRINT "MUST INCLUDE .BAS EXTENSION!":GOTO 350
440 IF F$="" GOTO 350
450 OPEN F$ FOR OUTPUT AS 1
460 RETURN
470 CLS
480 PRINT:PRINT
490 PRINT TAB(11)"********************";
500 COLOR 0,7
510 PRINT "  Visual Maker  ";
520 COLOR 7,0
530 PRINT "*********************"
540 PRINT TAB(11)"*                                                    * ";
550 PRINT TAB(11)"* Use the cursor pad arrow keys to move around screen. * ";
560 PRINT TAB(11)"* Press alphanumeric keys to type display.  You may    * ";
570 PRINT TAB(11)"* hit ESC, followed by a key to enter graphics mode.   * ";
580 PRINT TAB(11)"* In graphics, press any key other than arrow keys to  * ";
590 PRINT TAB(11)"* leave a trail of that graphics character.  Use arrow * ";
600 PRINT TAB(11)"* key to move without trail.  Exit graphics mode by    * ";
610 PRINT TAB(11)"* hitting ESC once again.                              * ";
```

```
620 PRINT TAB(11)"*                                                     * ";
630 PRINT TAB(11)"* Computer will BEEP when cursor reaches center of the * ";
640 PRINT TAB(11)"* screen.  Hit arrow keys once for each move; do NOT   * ";
650 PRINT TAB(11)"* hold arrow key down.                                 * ";
660 PRINT TAB(11)"*                                                      * ";
670 PRINT TAB(11)"* Press F1 to finish input.   You will be asked how    * ";
680 PRINT TAB(11)"* long you want each slide to be displayed.            * ";
690 PRINT TAB(11)"*                                                      * ";
700 PRINT TAB(11)"******************************************************* ";
710 PRINT TAB(26)"";
720 COLOR 16,7
730 PRINT " -- HIT ANY KEY TO BEGIN -- "
740 COLOR 7,0
750 GOSUB 230

755 ' *** Look for keyboard input ***

760 KEY(11) ON:KEY(12) ON:KEY(13) ON:KEY(14) ON
770 KEY(1) ON
780 CLS
790 GOSUB 230
800 A$=INKEY$:IF A$="" GOTO 800
810 IF A$<>CHR$(8) THEN GOTO 860
820 COL=COL-1:IF COL<1 THEN COL=1
830 LOCATE ROW,COL:PRINT CHR$(32);
840 CU=0
850 GOTO 910
860 IF A$=CHR$(27) AND FLAG2>0 THEN FLAG2=0:GOTO 920
870 IF A$=CHR$(27) THEN FLAG=1:GOTO 800
880 IF A$=CHR$(13) AND ROW<24 THEN ROW=ROW+1:COL=1
890 CU=ASC(A$)
900 IF FLAG=1 THEN CU=CU+128:FLAG=0:FLAG2=CU
910 IF COL=WIDE/2 THEN BEEP
920 LOCATE 25,1
930 COLOR 0,7
940 PRINT "Column: ";
950 COLOR 7,0
960 PRINT COL;
970 LOCATE 25,15
980 COLOR 0,7
990 PRINT "Row : ";
1000 COLOR 7,0
1010 PRINT ROW;
1020 LOCATE 25,25
1030 COLOR 0,7
1040 PRINT" Graphics : ";
1050 COLOR 23,0
1060 IF FLAG2>1 THEN PRINT " ON "; ELSE COLOR 7,0:PRINT "
      OFF"+SPACE$(18);
1070 COLOR 7,0
1080 IF FLAG2>1 THEN COLOR 0,7:PRINT "Character :";:COLOR
      7,0:LOCATE 25,55:PRINT CHR$(FLAG2);:COLOR 7,0
1090 LOCATE 25,58:COLOR 16,7:PRINT" F1-END  F10-Abort";:
      COLOR 7,0
1100 LOCATE ROW,COL
1110 IF VFLAG=1 THEN VFLAG=0:GOTO 1140
1120 IF FLAG2>0 THEN PRINT CHR$(FLAG2);:COL=COL+1:GOTO 1140
```

```
1130 IF CU>0 AND CU<>13 THEN PRINT CHR$(CU);:COL=COL+1
1140 IF CU=0 OR CU=13 THEN PRINT CHR$(43);
1150 CU=0
1160 GOTO 800

1165 ' *** Move Cursor ***

1170 LOCATE ROW,COL
1180 PRINT CHR$(32);
1190 COL=COL+1
1200 IF COL>WIDE-1 THEN COL=WIDE-1
1210 VFLAG=1
1220 RETURN 910
1230 LOCATE ROW,COL:PRINT CHR$(32);
1240 COL=COL-1
1250 IF COL<1 THEN COL=1
1260 VFLAG=1
1270 RETURN 910
1280 LOCATE ROW,COL:PRINT CHR$(32);
1290 VFLAG=1
1300 ROW=ROW+1
1310 IF ROW>24 THEN ROW=24
1320 RETURN 910
1330 LOCATE ROW,COL:PRINT CHR$(32);
1340 VFLAG=1
1350 ROW=ROW-1
1360 IF ROW<1 THEN ROW=1
1370 RETURN 910
1380 GOTO 790

1385 ' *** Check Screen Routine ***

1390 RETURN 1400
1400 GOSUB 1920
1410 IF SCREEN(ROW,COL)=43 THEN LOCATE ROW,COL:PRINT CHR$(32);
1420 LOCATE 25,1
1430 PRINT SPACE$(WIDE-1);
1440 LOCATE 25,10
1450 COLOR 16,7
1460 PRINT " Reading the Screen ";
1470 COLOR 7,0
1480 IF FFLAG=1 THEN GOTO 1530 ELSE FFLAG=1
1490 LN$(CU)=LN$(CU)+"KEY OFF"
1500 LN=LN+IC
1510 CU=CU+1
1520 GOSUB 1920
1530 LN$(CU)=LN$(CU)+"CLS"
1540 LN=LN+IC
1550 CU=CU+1
1560 :  FOR N=1 TO 24
1570 :    BFLAG=0
1580 :    EFLAG=0
1590 :    N3=0
1600 :    PR$=""
1610 :      FOR N1=1 TO WIDE
1620 :        N3=N3+1
1630 :        T=SCREEN(N,N1)
```

```
1640 :         LOCATE N,N1:PRINT CHR$(WHITE);
1650 :         IF BFLAG>0 THEN 1670
1660 :         IF T<>32 THEN BFLAG=N3: EFLAG=N3 ELSE 1690
1670 :         PR$=PR$+CHR$(T)
1680 :         IF T<>32 THEN EFLAG=N3
1690 :       NEXT N1
1700 :    IF RIGHT$(PR$, 1)=CHR$(32)THEN
           PR$=LEFT$(PR$,LEN(PR$)-1)
1710 :     IF EFLAG=WIDE THEN L$=";" ELSE L$=""
1720 :     IF BFLAG=0 THEN 1750
1730 :     LN$(CU)=STR$(LN)+" PRINT TAB("
            +STR$(BFLAG-1)+")"+CHR$(34)+MID$(PR$, 1,
            EFLAG-(BFLAG-2))+CHR$(34)+L$
1740 :     GOTO 1760
1750 :     LN$(CU)=STR$(LN)+" PRINT"
1760 :     CU=CU+1
1770 :     LN=LN+IC
1780 :  NEXT N
1790 CU=CU-1
1800 LOCATE 25,10
1810 LN=LN+IC
1820 PRINT"How many seconds should this frame be displayed";
1830 INPUT LENGTH$
1840 LN$(CU)=STR$(LN)+" F=TIMER+"+LENGTH$
1850 CU=CU+1
1860 LN=LN+IC
1870 LN$(CU)=STR$(LN)+" IF TIMER<F THEN GOTO "+STR$(LN)
1880 CU=CU+1
1890 LN=LN+IC
1900 LN$(CU)=STR$(LN)+" CLS"
1910 GOTO 1970
1920 LN=LN+IC
1930 CU=CU+1
1940 LN$(CU)=STR$(LN)+"  "
1950 RETURN

1955 ' *** Write to Disk ***

1960 CLS
1970 :  FOR N=1 TO CU
1980 :    PRINT #1,LN$(N)
1990 :    PRINT LN$(N)
2000 :  NEXT N
2010 DEF SEG=0
2020 POKE 1050,PEEK(1052)
2030 LOCATE 25,10
2040 PRINT SPACE$(50);
2050 LOCATE 25,21
2060 PRINT "Produce another frame?  ";
2070 COLOR 16,7
2080 PRINT "(Y/N)";
2090 COLOR 7,0
2100 A$=INKEY$:IF A$="" GOTO 2100
2110 IF A$="Y" OR A$="y" THEN ROW=1:COL=1:FLAG=0:
      FLAG2=0:GOTO 780
2120 CLOSE
2130 CLS
2140 END
```

Chapter 16



Word Processing Converter

Everyone agrees that standards are necessary to make personal computers really useful. As a result, there are dozens of different standards, not only between computers, but within a single computer line. As a result, you may find that your DOS 1.1 won't read 9-sector disks created by DOS 2.0 and greater, and one word processing program will use entirely different control codes than another.

The latter is a particular problem, because we often must trade text files with other IBM users. Your computer has 255 different character codes it recognizes (ignoring the extended codes for the moment). Only 52 of those are required for the upper and lowercase letters, and a dozen or so more for common punctuation, numbers, and other nongraphics characters. The ASCII codes for these have been fairly well standardized. (Although Commodore, for one, uses a different arrangement.) They occupy the codes from 32 to 128. Most manufacturers assign various graphic characters to the ASCII codes from 128 to 255, and many of the codes from 0 to 32 are devoted to agreed upon uses, such as CHR$(13) for carriage return, and CHR$(10) for linefeed.

Word processing programs usually need to indicate special conditions by a single character. The software author usually accomplishes this by assigning an ASCII code to that particular function. One may signify a page break; another might be a special end-of-paragraph marker. It is a common practice to insert *soft* carriage returns that can be eliminated by reformatting, as differentiated from *hard* carriage returns that mark the fixed end of lines.

There are no standards for these special codes, so software authors choose their own from the codes 128 to 255, or 0 to 32. Some WP programs use "escape" codes as well—these are a two-character code consisting of CHR$(27) plus some other character.

CONVERTING FOR COMPATIBILITY

Now, if you need to manipulate a WP file from one program with another, you may have terrible problems. At best, some of the control codes will be different and force you to make a lot of changes. At worst, none of them will match, and the text will be almost unreadable. Most WP programs have a nondocument mode (useful for editing BASIC programs, for example) that minimizes the differences. You might even be able to perform a global search and replace to substitute your WP program's control codes for those in the original file.

Better yet, use Converter, which will read a text file and convert control codes from one format to another. Converter has been set up so you can substitute the codes that apply to your particular word processing program and the one you most frequently convert from and to. If you have several, you can prepare a different version of the program for each.

HOW TO USE WORD PROCESSING CONVERTER

How do you determine what the relevant control codes are? The software manual may tell you. If not, I suggest running the following short program and writing down the CHR$ codes displayed when various appropriate points in the copy are reached.

```
100 LINE INPUT "ENTER
    FILENAME";F$
110 OPEN "I",1,F$
120 LINE INPUT#1,A$
130 FOR N=1 TO LEN(A$)
140 T$=MID$(A$,N,1)
150 PRINT ASC(T$);" ";T$;
160 B$=INPUT$(1)
170 NEXT N
180 GOTO 120
```

Each time you press a key, another character and its ASCII code will be displayed. Look for ends of paragraphs, possible page markers, and other codes. Write them down, find the equivalent for the other WP program (the same way) and then make the substitution in Converter.

The variables used in Converter are listed in Fig. 16-1. The file names of the two word-processor programs are defined in lines 120 and 130. Then the control codes for program A and program B are defined. The sample program includes page marker, carriage return, end of page, and soft carriage return. You can substitute control codes that best suit your application.

An array, CHARACTER$(row,col) is used to store these codes. The same routine can be used to convert either way, because FROM and INTO are defined in line 530, depending on the mode. The appropriate elements of CHARACTER$(row,col) are invoked during the conversion.

In line 570, one line of text is input into A$. Then a FOR-NEXT loop from 1 to 4 (change this if you have more than four codes to exchange) starts.

You need to find the first occurrence of ANY of the control codes in A$. The program looks for each in turn and stores in SMALLEST the position of the earliest. Variable N1 keeps track of which of the codes was the earliest one.

If a code is found, the leftmost portion of the program line is extracted up to the code, in line 630, and printed to the disk file. Then A$ is redefined as the remainder of the line, in line 640.

When the whole file is read, you are offered the opportunity to convert another. Converter is the least "finished" of any program in this book. You'll have to tailor it to your own word processing programs in order for it to work at all. By this time, however, you should have learned enough about handling ASCII files to make this chore a breeze. If not, go back to the beginning of the book and start reading again until you catch up with the rest of us. Go ahead. We'll wait.

A$	Used in INKEY$ loop.
FILEA$	File name A.
FILEB$	File name B.
FROM	Which mode is to be converted from.
G	Location in string of any code.
L$	Left portion of string up to code.
SMALLEST	First appearance of any of the codes to be converted.
TO	Which mode is to be converted to.

Fig. 16-1. Variables used in Converter.

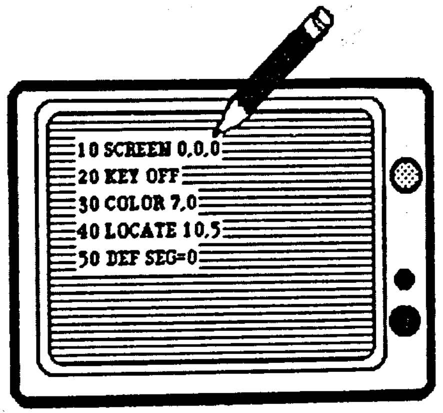
Listing 16: The Converter Program

```
10 ' **************
20 ' *            *
30 ' *  Converter *
40 ' *            *
50 ' **************
60 '

65 ' *** Initialize ***

70 SCREEN 0,0,0
80 KEY OFF
90 COLOR 7,0
100 ON KEY(10) GOSUB 750
110 KEY(10) ON
120 FILEA$="Program A"
130 FILEB$="Program B"
140 CLS
150 LOCATE 25,30
160 COLOR 16,7
170 PRINT" Hit F10 to abort. ";
180 COLOR 7,0
190 LOCATE 4,8
200 PRINT "== WP File Translation Utility =="
210 PRINT:PRINT
220 PRINT TAB(12)" By: David D. Busch
230 PRINT:PRINT

235 ' *** Define codes to exchange ***

240 PAGE.MARKER.A$=CHR$(12)
250 PAGE.MARKER.B$=CHR$(142)
```

```
260 CARRIAGE.RETURN.A$=CHR$(27)+CHR$(69)
270 CARRIAGE.RETURN.B$=CHR$(13)
280 END.OF.PAGE.A$=CHR$(27)+CHR$(69)+CHR$(27)+CHR$(71)
290 END.OF.PAGE.B$=CHR$(141)
300 SOFT.RETURN.A$=CHR$(27)+CHR$(70)
310 SOFT.RETURN.B$=CHR$(4)
320 CHARACTER$(1,1)=PAGE.MARKER.A$
330 CHARACTER$(1,2)=PAGE.MARKER.B$
340 CHARACTER$(2,1)=CARRIAGE.RETURN.A$
350 CHARACTER$(2,2)=CARRIAGE.RETURN.B$
360 CHARACTER$(3,1)=END.OF.PAGE.A$
370 CHARACTER$(3,2)=END.OF.PAGE.B$
380 CHARACTER$(4,1)=SOFT.RETURN.A$
390 CHARACTER$(4,2)=SOFT.RETURN.B$

395 ' *** Enter filename ***

400 PRINT TAB(8)"Enter name of file to process :"
410 PRINT TAB(8)"";
420 LINE INPUT F$
430 PRINT TAB(8)"Enter name of output file: "
440 PRINT TAB(8)"";
450 LINE INPUT F2$

455 ' *** Set Mode ***

460 PRINT:PRINT
470 PRINT TAB(4)"Do you want to:"
480 PRINT TAB(6)"1.) Convert from ";FILEA$;" to ";FILEB$;"
      format"
490 PRINT TAB(6)"2.) Convert from "FILEB$;" to
      ";FILEA$;"format"
500 A$=INKEY$:IF A$="" GOTO 500
510 A=VAL(A$)
520 IF A<1 OR A>2 GOTO 500
530 IF A=1 THEN FROM=1:INTO=2 ELSE FROM=2:INTO=1

535 ' *** Open Disk files ***

540 OPEN "I",1,F$
550 OPEN "O",2,F2$
560 IF EOF(1) GOTO 690

565 ' *** Load a line ***

570 LINE INPUT#1,A$
580 FOR N=1 TO 4
```

```
590 G=INSTR(A$,CHARACTER$(N,FROM))
600 IF G<>0 AND G<SMALLEST THEN SMALLEST=G:N1=N
610 NEXT N
620 IF G=0 THEN GOTO 670
630 L$=LEFT$(A$,SMALLEST-1)+CHARACTER$(N1,INTO)
640 A$=MID$(A$,SMALLEST+1)
650 PRINT #2,L$;
660 GOTO 580
670 PRINT #2,A$;
680 GOTO 560
690 CLOSE

695 ' *** Do again? ***

700 CLS
710 LOCATE 25,10
720 PRINT"Process another file?";
730 A$=INKEY$:IF A$="" THEN GOTO 730
740 IF A$="Y" OR A$="y" THEN RUN
750 CLS
760 CLOSE
770 END
```

Chapter 17



Unpacker

Unpacker is the last demonstration of ways to manipulate ASCII files. The final program in the book, Music Writer, will create files, but not edit them. This program will read in your ASCII format program files and, where possible, rewrite them so that each statement is on a separate line. This may make debugging easier and the program a bit simpler to understand. It has some limitations, but they are few.

By now you should understand how Unpacker works even without any explanation. The concept is simple enough to be explained in a few sentences. The program reads in each program line, as we have done previously. It looks for colons, which separate statements. If a colon is found, the line is broken at that point, and the remainder of the program line is assigned a new line number and printed as the next line. Colons inside quotation marks are ignored. There is no provision to allow for colons after REMarks, however, so you should use some caution. Figure 17-1 lists the variables used in Unpacker.

Line 350 sets P, the variable that indicates the position at which the search will begin, to one. Then, as each program line is read in, starting at line 360, the program looks for the first space in the line, S, the first occurrence of the reserved word "IF", and the position of a colon. The first and second appearances of quotation marks are also noted.

If you happen to have left off a closing quotation mark, the rest of the program line will be considered to be within the quote by the computer. In this case, the position of the missing quote is set as the length of the program line.

Next, the Unpacker looks to see if G, the position of the colon, is greater than that of the first quotation mark and less than that of the second, meaning that it is inside the quotation marks. The position of "IF" is also examined to make sure it is not inside quotation marks. If either condition is true, then P is set to the position after the second quote, and the program loops back to continue the search.

If the colon or "IF" are not within quotation

A$	Line input from the file.
F$	File being processed.
F2$	Output filename.
F	Location of "IF".
G	Location of colon.
LN	Line number.
Q1,Q2	Location of quotation marks.
S	Location of first space in the remaining line.

Fig. 17-1. Variables used in Unpacker.

marks then the program goes on to process the line. As you know, when the PC encounters an IF statement, the rest of the program line is carried out only if the statement that follows IF is true, except where an ELSE is provided. In fact, there may be nested IFs and ELSEs that can truly make the logic difficult to follow. In fact, this is to much for a simple program like Unpacker. We don't want to mess up true statements that follow "IF," so when IF is found, the program stops dividing up the line and continues to the next. In other words, it avoids the problem by skipping it altogether. This is a time honored programming practice that is frowned upon. If however, we can achieve 90 percent of the desired goals of Unpacker without going through contortions, it may be worth it to bend the rules a bit. Points of diminishing returns CAN be reached even in programming.

If there is no IF, the line number of the current line is calculated. Since the first characters on a line, up to the first space, will always be the line number, the program can find the line number by taking LEFT$ (A$,S-1).

LN$ is defined as everything from the beginning of the line up to the colon (minus one). LN$ is printed to the disk file in line 580. A$ is redefined as everything following the colon. A new line number is needed for A$, so LN+1 is used. Then the program goes back to look at the new A$ for additional colons. That's all there is to this simple but useful utility program.

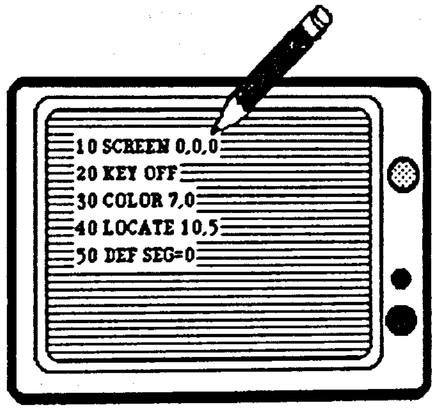
Listing 17: The Unpacker Program

```
10 ' **************
20 ' *            *
30 ' *  Unpacker  *
40 ' *            *
50 ' **************
60 '

65 ' *** Initialize ***

70 SCREEN 0,0,0
80 KEY OFF
90 ON KEY(10) GOSUB 700
```

```
100 KEY(10) ON
110 COLOR 7,0
120 CLS:PRINT:PRINT
130 LOCATE 25,30
140 COLOR 16,7
150 PRINT" Hit F10 to abort. ";
160 COLOR 7,0
170 LOCATE 4,12
180 COLOR 0,7
190 PRINT "== Un Packer Utility =="
200 COLOR 7,0
210 PRINT:PRINT
220 PRINT TAB(12)" By: David D. Busch"

225 ' *** Enter filename to process ***

230 PRINT:PRINT
240 PRINT
250 PRINT TAB(8)"Enter name of file to process :"
260 PRINT TAB(8)"";
270 LINE INPUT F$
280 PRINT TAB(8)"Enter name of output file :"
290 PRINT TAB(8)"";
300 LINE INPUT F2$

305 ' *** Open disk files ***

310 PRINT:PRINT
320 OPEN "I",1,F$
330 OPEN "O",2,F2$

335 ' *** Start new line ***

340 IF EOF(1) GOTO 640
350 P=1
360 LINE INPUT#1,A$

365 ' *** Look for spaces, IF, and colons ***

370 S=INSTR(P,A$,CHR$(32))
380 F=INSTR(P,A$,"IF ")
390 G=INSTR(P,A$,":")
400 IF S<>0 AND G>S THEN GOTO 410 ELSE GOTO 480
410 T$=MID$(A$,S,G-S)
420 FOR N2=1 TO LEN(T$)
430 IF MID$(T$,N2,1)<>CHR$(32) THEN SFLAG=1
440 NEXT N2
```

```
450 IF SFLAG=1 THEN SFLAG=0:GOTO 480
460 A$=LEFT$(A$,S)+MID$(A$,G+LEN(T$))
470 SFLAG=0:GOTO 370

475 ' *** Find Quotes ***

480 Q1=INSTR(P,A$,CHR$(34))
490 Q2=INSTR(Q1+1,A$,CHR$(34))
500 IF Q1>0 AND Q2=0 THEN Q2=LEN(A$)
510 IF G=0 THEN GOTO 610
520 IF G>Q1 AND G<Q2 THEN P=Q2+1:GOTO 370
530 IF F>Q1 AND F<Q2 THEN P=Q2+1:GOTO 370
540 IF F>0 AND F<G THEN GOTO 610
550 LN=VAL(LEFT$(A$,S-1))
560 LN$=LEFT$(A$,G-1)
570 A$=MID$(STR$(LN+1),2)+" "+MID$(A$,G+1)

575 ' *** Write to Disk ***

580 PRINT #2,LN$
590 PRINT LN$
600 GOTO 370
610 PRINT #2,A$
620 PRINT A$
630 GOTO 340
640 CLOSE

645 ' *** Do again? ***

650 CLS
660 LOCATE 25,10
670 PRINT"Process another file?    (Y/N)"
680 A$=INKEY$:IF A$="" GOTO 680
690 IF A$="Y" OR A$="y" THEN RUN
700 CLS
710 CLOSE
720 END
```

Chapter 18



Creating Your Own DOS Commands

Most of this book has been concerned with BASIC programming tips and utilities. However, I've mentioned some of the interesting things you can do with PC-DOS, and it might be fun to slip in a few of them for you to play with. For example, wouldn't you like to create your own DOS commands?

I, for one, have not yet gotten accustomed to the PC's keyboard, and typing in DIR B: can be fraught with confusion as I try to remember to hit the Shift key to get the colon. I don't even bother anymore. When I want to see the directory of drive B:, I just type D B. If I happen to be logged onto B:, I can just hit D.

Some of us are fair spellers, but have difficulty remembering acronyms and abbreviations. Is it "CHKDSK", or "CHCKDSK" or what? No bother. With my computer system, I just type CHECK A or CHECK B to examine the desired drive.

This magic is worked through BATCH files. These are system files, nothing more than ASCII text, with the .BAT extension. When you invoke a batch file, the IBM PC will look at each line and attempt to execute it as if it were entered from the keyboard. On powerup, the PC will look for a special batch file, AUTOEXEC.BAT. If it finds it, those commands will be executed automatically, without your needing to do anything.

ALTERING THE SYSTEM PROMPT

AUTOEXEC.BAT is a good way to custom-configure your system the way you want it. You can run utilities that set the system clock to a clock board you've installed, activate a RAM drive, or do other tasks on powerup. Here's a line that is in my own AUTOEXEC.BAT file:

```
PROMPT $t$h$h$h$h$h$h$_$n$g
```

A bit cryptic, right? PROMPT lets you alter the system prompt, using several special characters, each preceded by a dollar sign to differentiate the special characters from any other string you might want to include in the prompt. Here are the special characters that are legal:

t	the time

d	the date
p	the directory of the default drive
v	the DOS version number
n	the default drive name
g	the greater than symbol
l	the less than symbol
b	a blank space
q	the equals sign
h	backspace
e	ESCAPE
-	Go to the next line on the screen

So, typing PROMPT ng would set the system prompt to the default drive name and the greater than symbol, like this:

```
A>  or B>
```

That is the normal prompt setting. You can change the prompt to include the time, date, DOS version number, and other information as you want. Perhaps you have deciphered my own system prompt shown above. It looks something like this, as a two-line prompt:

```
22:37
A>
```

I include the time, $t, followed by six backspaces, $h, so the seconds and fraction are written over. I care only about whole minutes. Then the prompt drops down a line and prints the normal default drive and ">" information. When I need to know the time, I simply press the Return key, and my system prompt tells me. The rest of the time the clock ticks away unobtrusively. As a side benefit, I can tell at a glance how long it has been since I used my PC. I press the Return key and compare with the system prompt above it.

SEARCHING THROUGH DISK DRIVE DIRECTORIES

There is one very important line you should include in your AUTOEXEC.BAT file, especially if you want to define your own DOS commands. That line looks something like this:

```
PATH A:\;B:\
```

That command, once invoked, will cause the system to search through the directories of your disk drives in that order when it cannot locate a batch file or command in the currently logged directory.

I repeat: the PATH command can tell DOS to look on other disk drives besides the currently logged disk for a batch file or command.

Do you understand what that means? If you have tried to load BASIC, which is stored on A: when you happen to be logged to B:, you probably have wished that DOS were smart enough to go look on a different drive if it couldn't locate a file. Well now, at least with command files and batch files, you can tell DOS to do that very thing! It makes it practical to use batch files as new DOS commands, because it does not matter where you happen to be logged when you decide to use a command. The command will be faster if it is located on the logged drive, but it will work on any drive that you have specified with the PATH command.

CREATING BATCH FILES

Now, on to the batch files themselves. When you type a file name with no extension, DOS first looks to see if there is a .COM or .EXE file with that name. Then it checks to see if there is a .BAT file that matches. If so, it will execute that batch file. If you wanted to invent a command called "CHECK," which would invoke CHKDSK, you could create a batch file called CHECK.BAT with the single line: CHKDSK. Then, typing CHECK would summon CHKDSK automatically.

To write your own batch file, just copy from the console. Here is a sample session:

```
COPY CON:CHECK.BAT<ENTER>
```

```
CHKDSK<F6>
(1) files copied.
```

Using F6 instead of the Enter key ends your batch file input, while saving you a carriage return in the file. Now, the batch file you have just created is useful; however you can make it more so. DOS allows you to specify up to 10 parameters on the same line as the command invoking the batch file. These parameters will be dropped into the batch file in the locations indicated by numbers you put there, "%0", "%1", "%2", "%3", and so forth. They will be included in the order you place them on the command line, but they do not have to be in the same order in the batch file. Try this line in your CHECK batch file:

```
CHKDSK %1: /F
```

Now, from DOS you type:

```
CHECK B
```

When the batch file is executed, DOS substitutes the B parameter for the %1, and the command is now:

```
CHKDSK B: /F
```

CHKDSK will do a check of drive B: plus fix any lost data, as directed by the /F switch. That's all there is to creating that new DOS command.

Try this one:

```
COPY CON:D.BAT
DIR %1: /W
```

Now, you can invoke D.BAT by typing:

```
D A or D B
```

and get the directory of the drive you want. The /W will display the directory in the wide format. You could substitute /P and have the directory shown in columns, but pausing when the screen is filled.

CREATING A TEXT FILE

Having difficulty remembering the syntax for certain DOS internal or external commands, like the MORE filter? We can't have a batch file with the same name as a .COM or .EXE file, so I have created a file that shows me, page by page, a text file, from DOS, using MORE. I call it LOOK.BAT. This one needs two parameters, one for the file to be looked at and one for the drive on which it resides:

```
COPY CON:LOOK.BAT
MORE<%2:%1
```

Now, I type LOOK MYFILE.TXT B, when I want to look at MYFILE, which is on drive B:. DOS substitutes, coming up with:

```
MORE<B:MYFILE.TXT
```

This happens to be the syntax, which I never remember, to use the MORE screen display filter on MYFILE.TXT, so I can look at it a screen at a time, with the —MORE— pause in between pages. It's slick.

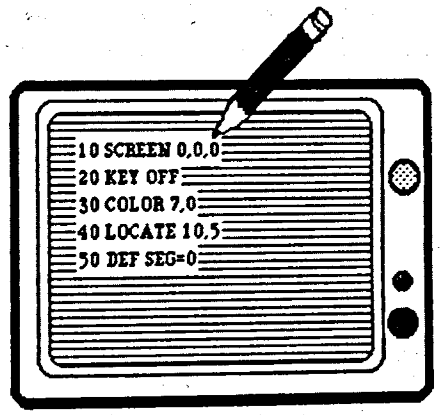
MORE WAYS TO USE BATCH FILES

Think of any command you'd like DOS to have. When I am on drive A:, but I'd like to be logged onto drive B:, and in BASIC, I just type BASICB, and guess what happens? You probably can create a batch file to implement commands of your own. Keep in mind the rules: no conflicts in batch file names between existing commands. Only 10 parameters, numbered %0 through %9, can be used, unless you use SHIFT, a batch file subcommand. That's a bit complicated and not particularly useful since more than 10 parameters are unwieldy.

You might, however, be interested in constructing your batch files and DOS commands using the other batch subcommands available. These include PAUSE, which stops the batch file until you press a key, REM, which allows you to embed remarks, and BREAK, which tells DOS to look for a control break whenever a program asks for DOS functions.

You can also make batch files into little programs on their own with IF, GOTO, and FOR subcommands. This chapter is not intended to be a complete tutorial on batch files. But I hope I've gotten you interested in finding out more ways to use them to make your Automatic IBM PC operate more efficiently for you. Just keep in mind that the computer can do anything you tell it to do, and the IBM is supplied with modes of instructions second to none.

Chapter 19



Music Writer

I have one more program for you. Music Writer will write programs that play songs! All you have to do is enter the names of the notes and how long you want them played. The program will write lines that, when run, will play the song you have entered.

Music Writer uses BASIC's PLAY command, which can play strings of notes through the PC's speaker. Unlike the SOUND command, which requires that you enter the frequencies of the notes:

```
10 SOUND 440,10
```

PLAY lets you enter the actual note names:

```
10 PLAY "A#BCD#"
```

or

```
10 F$="BCDEF"
20 PLAY F$
```

As you might have noted, sharped notes are indicated by following the letter name with a # (or +, since sharped notes are half a tone higher.) Flats are indicated with a minus sign. That is, B – would be B-flat. Music students will know that only half a tone separates some notes, so in those cases flats and sharps are not allowed. For example, B and C are only a half-tone apart, so B# and C-flat are illegal. The IBM PC is smart enough to know this.

Your PLAY string can include other characters in addition to the notes A to G. If you include "O", followed by a number 0 to 6, an octave will be chosen. Each octave goes from C to B. AN "L", followed by a number from 1 to 64 will indicate the length of a note, with 1 being a whole note, 2 a half note, on up to 64, (a 64th note, or 1/64th of a beat.) A "P" (for pause) can be used with the same numbers to produce a rest, or silence, of the indicated length.

Using a "T" in your string, accompanied by a number from 32 to 255, will set the tempo, in quarter notes per minute. The default is 120.

You can also include several other strings, such as "ML" for music legato or "MS" for music staccato. You really need to know your music to use these correctly. Consult the IBM BASIC guide for tips on using these commands.

A typical string might look like this:

```
10 F$="L8ACDEP2G#DL16ACDEO4"
```

Now, you can sit down at your PC and write these strings, using sheet music if you wish. It is, however, easy to make a mistake, and writing a program to play the strings can be time consuming. The Automatic PC can do it for you.

Music Writer will let you enter strings and perform some error checking to make sure that each "L" or "P" is always followed by a number in the range 1-64 and that numbers don't appear where they don't belong. Only the correct notes will be allowed, with C-flat automatically filtered out.

Pressing F1 will stop the programming at any time. As always, you can abort by pressing F10.

The program, the variables in which are shown in Fig. 19-1, works like this: the file name for the output file is entered, and the file opened. Then a string, F$, which includes the notes and characters that can be input, is defined.

The last note entered, at location 5,5 (row 5, column 5) is erased from the screen. Then an INKEY$ loop starts to wait for you to press a key. If the key pressed is backspace (CHR$(8)), and notes have been entered, the rightmost character is deleted from NOTE$, which stores the notes entered so far.

If E$ equals carriage return (CHR$(13)), the program begins processing the string you have entered. The first step is to change any lowercase letters to uppercase. Then the ASCII value of the first character in A$, and the VAL are taken. The LAST character entered, R$, is also found, so the program can see if a "P," "L," or "T" was entered. If so, it insists that the next entry be a number in the proper range. If wrong entries are made, the program branches to various error routines. When NOTE$ becomes longer than 200 characters, or if you press F1, the program writes the NOTE$ to disk, building a PLAY program line, in a manner similar to the way lines are built in many other programs in this book. At the same time, the program PLAYS the NOTE$ you have compiled.

You'll find that Music Writer gives you a fast way to key in your favorite tunes, while keeping you from making many input errors. In fact, I've carried the error trapping almost to extremes. After you press each key, the program will prompt you as to what type of input is expected next. If the octave you've chosen is too high, it will tell you that. If the tempo is too slow or fast, you'll be notified. The proper input format is displayed on the screen at all times.

In short, you should press the Enter key after each note is completed. For example, if you want the notes, A, B-flat, C, you would type A<ENTER>, B-<ENTER, and C<ENTER>. As soon as you type in the A, the program will tell you that the next character must be a plus, minus, or # or Enter. It doesn't check to see if the note is a legal one, e.g., B-flat, until you press the Enter key, however.

When you type a letter such as O or T or P or L, which must be followed by a number value, the program will immediately add the character to the string, without your having to press enter. Then you will be told that the next entry must be a number. You can always backspace to correct a note or other character entered in error. If you backspace to correct an entry and go back as far as one of those letters, then your next entry must again be a number. I've made Music Writer as foolproof as possible. It's almost impossible to make an illegal entry. Incorrect entries are still within the realm of possibility. So I haven't removed all the fun for you. If you wish, you can compose some awful-sounding music—just more efficiently.

A$	String entered by user.
COL	Column to print string.
DELAY	Delay loop counter.
E$	Used in INKEY$ loop.
F$	Allowable characters.
LN$	Current line number of program being written.
MUSIC$	Filename of program being written.
N	Loop counter.
R$	Last character entered.
ROW	Row to print string.
U	ASCII value of first character in A$.

Fig. 19-1. Variables used in Music Writer.

Listing 19: The Music Writer Program

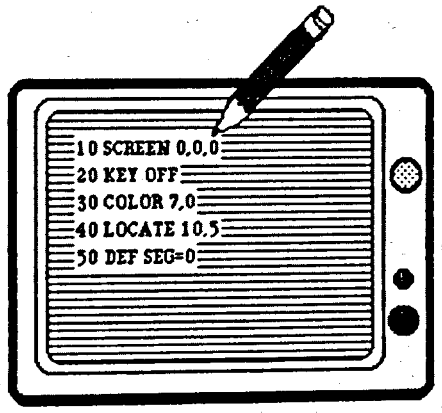
```
10 ' ****************
20 ' *              *
30 ' * MUSIC WRITER *
40 ' *              *
50 ' ****************

55 ' *** Initialize ***

60 CLS
70 SCREEN 0,0,0
80 COLOR 7,0
90 KEY OFF
100 ON KEY(1) GOSUB 930
110 ON KEY(10) GOSUB 980
120 KEY(1) ON
130 KEY(10) ON
140 LOCATE 10,1
150 PRINT SPACE$(65);
160 LOCATE 10,1

165 ' *** Enter name for output file ***

170 BEEP
180 PRINT"ENTER FILENAME FOR MUSIC :";
190 LINE INPUT F$
200 FOR N=1 TO LEN(F$)
210 T=ASC(MID$(F$,N,1))
```

```
220 IF T>96 AND T<123 THEN MID$(F$,N,1)=CHR$(T-32)
230 NEXT N
240 IF LEN(F$)>12 THEN PRINT"File name too
    long!":PRINT:GOTO 140
250 S9=INSTR(F$,".BAS")
260 IF LEN(LEFT$(F$,S9))>8 THEN PRINT"File name too
    long!":PRINT:GOTO 140
270 IF S9=0 THEN PRINT "MUST INCLUDE .BAS EXTENSION!":
    GOTO 140
280 IF F$="" GOTO 140
290 OPEN "O",1,F$
300 GOTO 370
310 LOCATE 25,55
320 COLOR 16,7
330 PRINT "F1-QUIT SONG  F10-ABORT";
340 COLOR 7,0
350 RETURN
360 FOR N=1 TO 500:NEXT N
370 CLS:GOSUB 310
380 LOCATE 2,10
390 PRINT"START ENTRY NOW :"
400 F$="A-A#A+B-C+C#D-D#D+E-F#F+G-G#G+OLPTMX<>MFMBMNMLMSX"
410 ROW=10:COL=5
420 LOCATE 5,5
430 PRINT SPACE$(70)
440 GOSUB 1010
450 LOCATE 5,5
460 X=CSRLIN:Y=POS(Z)

465 ' *** Wait for input ***

470 E$=INKEY$:IF E$="" GOTO 470

475 ' *** Handle backspace ***

480 IF E$=CHR$(8) AND LEN(NOTE$)<1 THEN GOTO 470
490 IF E$=CHR$(8) THEN LFLAG=0:LOCATE ROW,COL:PRINT
    SPACE$(LEN(NOTE$)):NOTE$=LEFT$(NOTE$,(LEN(NOTE$)-1)):
    G$=RIGHT$(NOTE$,1):IF G$="T" OR G$="P" OR G$="L" OR
    G$="O" THEN LFLAG=1:GOTO 810 ELSE GOTO 810

495 ' *** Check for required number ***

500 IF LFLAG=1 AND VAL(E$)<1 THEN LOCATE 25,1:
    PRINT"A Number Please!!!   ";:BEEP:
    FOR N1=1 TO 800:NEXT N1:GOTO 900
510 IF LFLAG=1 AND VAL(E$)>0 THEN LFLAG=0
```

```
515 ' *** Check for required sharp or flat or natural ***

520 IF E$="-" OR E$="+" OR E$=CHR$(13) OR E$="#" THEN
    LOCATE 25,1:PRINT SPACE$(50); ELSE IF NFLAG=1 THEN
    BEEP:LOCATE X,Y:GOTO 470
530 LOCATE X,Y
540 A=ASC(E$):IF A>96 AND A<123 THEN A=A-32:E$=CHR$(A)
550 X=CSRLIN:Y=POS(Z)+1

555 ' *** Prompt for a number ***

560 IF E$="P" OR E$="T" OR E$="O" OR E$="L" THEN LOCATE
    25,1:PRINT SPACE$(50);:LOCATE 25,1:COLOR 0,7:PRINT"
    Now enter a number. ";:COLOR 7,0:LFLAG=1
570 LOCATE X,Y
580 PRINT E$;:IF LFLAG=1 THEN A$=E$:E$=CHR$(13)

585 ' *** Prompt for sharp, flat or natural ***

590 IF INSTR("ABCDEFG",E$)<>0 THEN LOCATE 25,1:COLOR
    0,7:PRINT"Now # OR + (sharp), - (flat), or
    <ENTER>(natural)";:COLOR 7,0:LOCATE X,Y:NFLAG=1
600 IF E$<>CHR$(13) THEN A$=A$+E$:GOTO 470
610 IF A$="" GOTO 420
620 U=ASC(A$)
630 NUMBER=VAL(A$)
640 IF NUMBER>0 OR R$="O" THEN GOTO 660
650 IF INSTR(F$,A$)=0 THEN GOTO 900
660 R$=RIGHT$(NOTE$,1)
670 IF NUMBER<7 AND R$="O" THEN GOTO 790
680 IF R$="P" OR R$="L" THEN GOTO 690 ELSE GOTO 730
690 IF RIGHT$(NOTE$,2)="ML" THEN GOTO 730
700 IF NUMBER>64 THEN GOTO 870
710 IF NUMBER<1 THEN GOTO 900
720 GOTO 790
730 IF NUMBER<7 AND R$="O" THEN GOTO 790
740 IF NUMBER>31 AND NUMBER<256 AND R$="T" THEN GOTO 790
750 IF R$="T" THEN GOTO 880
760 IF R$<>"O" THEN GOTO 780
770 IF U<47 OR U>54 THEN GOTO 890
780 IF U>46 AND U<57 THEN GOTO 900
790 IF A$="-" OR A$="+" OR A$="#" THEN
    A$=R$+A$:NOTE$=LEFT$(NOTE$,(LEN(NOTE$)-1)):GOTO 770
800 NOTE$=NOTE$+A$
810 LOCATE ROW,COL
820 PRINT NOTE$
830 IF LEN(NOTE$)>200 THEN GOTO 910
```

```
840 A$=""
850 NFLAG=0
860 GOTO 420

865 ' *** Notify of errors ***

870 LOCATE 25,1:PRINT "NOTE OR REST TOO LONG";:GOTO 900
880 LOCATE 25,1:PRINT"TEMPO INCORRECT";:GOTO 900
890 LOCATE 25,1:PRINT"OCTAVE WRONG";:GOTO 900
900 LOCATE 25,25:COLOR 0,7:PRINT" INVALID CHOICE
    ";:BEEP:FOR DELAY=1 TO 800:NEXT DELAY:LOCATE 25,1:COLOR
    7,0:PRINT SPACE$(50);:A$="":GOTO 420
910 GOSUB 930
920 GOTO 420

925 ' *** Write song to disk ***

930 LN=LN+10
940 PLAY NOTE$
950 LN$=STR$(LN)+"  "+"PLAY "+CHR$(34)+NOTE$+CHR$(34)
960 PRINT #1,LN$
970 NOTE$=""
980 CLOSE
990 CLS
1000 END

1005 ' *** Show entry style ***

1010 LOCATE 3,30
1020 PRINT"Enter in following style:"
1030 LOCATE 5,30
1040 PRINT"Note names:  B- <ENTER>"
1050 LOCATE 6,30
1060 PRINT"Pauses:      P<ENTER> 2<ENTER>"
1070 LOCATE 7,30
1080 PRINT"Length:      L<ENTER> 2<ENTER>"
1090 LOCATE 8,30
1100 PRINT"Octaves:     O<ENTER> 1<ENTER>"
1110 LOCATE 9,30
1120 PRINT"Tempo:       T<ENTER> 60<ENTER>"
1130 RETURN
```

Chapter 20



Some Tips

The whole aim of this book has been to show you how to make your programming more efficient by letting other programs write your code for you. The sixteen programs presented so far generate program lines, modify software, or perform other tasks for you. But there is no reason to limit your automatic IBM PC to just those utilities included here. Actually, there are many, many programs on the market that will streamline your work.

DEVELOPING A PROGRAM WITH A WORD PROCESSOR

There is one tool you may not have thought of, unless you are an old time programmer, write in assembly language, or write for compilers. That utility is your word processor. Word processors of today have much in common with text editors that have been used in the past to write programs that are compiled or assembled into machine language code. Most Basic programmers today, however, have never written a program with a text processor. The majority have worked only with interpreters. An interpreter is, of course, a computer program that takes the instructions written by the programmer and translates them into the computer's machine language each time a line is run.

That is, when a line like FOR N=1 TO 50:B=A+C:NEXT N is encountered, the interpreter will calculate the machine code fifty different times. This is why interpreters are so much slower than machine language programs. There are, however, advantages to interpreters. One is that a program can be written a small part at a time, and each section run, tested, and then modified immediately. Another advantage is that interpreters can include error trapping features that handle improper user input—such as attempts to store numbers larger than 32767 in an integer variable—that might have been unanticipated when the program was written.

Compilers and assemblers are less forgiving. Code is written, and the source code used to produce the run-time object code. Mistakes can only be corrected by modifying the source code and compiling or assembling new object code. Partially because of this, BASIC interpreters have been the favored program development tool. And, IBM PC

BASIC programmers have missed some of the editing and program writing tools possible with word processors.

Of course, the PC has both screen and line editing. It's nice to be able to move the cursor around and change code rapidly. Those of us accustomed to word processing, however, appreciate other features, such as global searching and replacing, and zipping from one portion of a document to another.

But wait. What if the program were loaded into the word processor as if it were a document? The arrow keys could be used to zip the cursor around the program, and changes made by overtyping, global search and replace, and other powerful features.

The only "trick" to using a word processing program as a program editor is to remember to save the program from BASIC in ASCII form. Then it usually can be loaded into the word processing program. You must also take care to store the program from the WP software in ASCII or *nondocument* mode, as well. If you forget this step and attempt to load the program, only a few characters of garbage will appear on the screen. Don't panic. Return to the word processing program, reload the compressed program file, and then re-SAVE it in ASCII.

What can you do with a program in text form? For starters, how about formatted listings even slicker than those produced by LISTER? The latter was provided both as an illustration and for those who do not have a WP program; however, a word processing program was used to print out the listings reproduced in this book. The word processing software divided up the program lines into pages and printed a header at the top of each page.

My WP program allows setting the window of the IBM PC's screen to the same width as the paper being used, so it was simple to scroll down through the program text to see when lines were too long. In most programs, for clarity, line breaks were chosen and the next part of a line indented. A word processing program was also used to add spacing between REMarks and the program lines preceding and following.

Although original code cannot be tested while in a WP program, there are advantages that make them very desirable. Here are a few tips for using a word processing program to streamline your program writing. Those of you with other WP programs can use them as well, by applying the particular syntax and commands of your favored text processor.

☐ Put your most-used modules at the tips of your fingers. Several phrases and program lines were written and encased in blocks given unique markers. If your WP software does not allow marking multiple blocks, perhaps you can store these phrases in the Library or boilerplate file. Then, when a phrase like A$=INKEY$:IF A$="" GOTO was needed, it was a simple matter to install it from the built-in library of routines.

Of course, it would have been simpler to write subroutines and call these rather than write the code over and over, even automatically. But, "easier" is not always as clear for someone attempting to understand a BASIC program, so in many cases, subroutines were avoided. Programming speed did not slow down, however, because of the power of the word processing program.

☐ Global searches, replaces, and deletions made writing the programs in this book much easier, as well. Halfway through a program, on discovering that a variable name was ill-chosen, it was a simple matter to replace all occurrences in a couple seconds. REM *** could be changed to ' *** almost instantly. Some program screens, written using Screen Editor, had PRINT TAB(0) in a number of places. All the TAB(0) appearances could be deleted quickly.

Care has to be taken when using this feature, however. A word processor will not check to see whether or not the string being changed is inside quotation marks. Changing all PRINTs to LPRINTs can result in some undesired modifications, such as LPRINT becoming LLPRINT, or "IS YOUR PRINTER ON?" being transformed into "IS YOUR LPRINTER ON?"

☐ Programs can be "cleaned up" quite easily. It is fast and efficient to zip through a program with a word processor and touch up sloppy coding, change all-uppercase prompts to upper and lowercase, or delete undesired spaces. After writing TABBER, I wanted to go through some earlier programs and center prompts. Unfortunately some program lines had prompts with, horrors, embedded spaces:

```
10 PRINT " DO YOU WANT TO:"
20 PRINT "          1.) RUN A
                        PROGRAM"
30 PRINT "          2.) EXIT THIS
                        PROGRAM"
40 PRINT "             ENTER CHOICE:"
```

While it was easy to type like that when writing the original program, someone typing in the program from this book would be hard pressed to count the number of spaces needed to properly format the lines on the screen. By replacing all PRINT " with PRINT" one space was closed up. Then by replacing all PRINT"   with simple PRINT statements and quotation marks, the excess spaces inside the prompts were eliminated. Then the PRINT TAB(T)'s could be put where needed, and TABBER could be used successfully.

PROTECTING YOUR WORK

Here's another quick tip. As a program is developed, it is good practice to save the work in progress to disk periodically. Thus, should a power failure occur, hours' worth of work is not lost.

With disk-based systems, backups are very easy—so simple, in fact, that many programmers end a session, look at the working disk's directory, and see 10 or more versions tucked away—PROGRAM1.BAS PROGRAM2.BAS, and so on. This system works fine, but few of us can remember what we called the last version when we are ready to save the next version. Either we invoke FILES to check, or play it safe and skip a number or two.

Here's a short program that can be appended onto the end of any program you are working on and used to automatically SAVE an updated version of the program, under an appropriate file name. When you type GOTO 30000 at any point during program development, the module will collect the current TIME$, extract the hour and minutes, and use that to make the file name.

```
30000 B$=TIME$:H$=MID$(B$,10,2)
30010 M$=MID$(B$,13,2)
30020 F$="PROG"+H$+M$+.BAS
30030 SAVE F$
```

Save these lines in ASCII form on your disk, and then APPEND or MERGE it to any program you choose (which does not have line numbers that conflict). You may want to EDIT line 30020, replacing the string "PROG" with any four letters that are more meaningful for the program you are developing. If you want to back up the program to two (or more) disk drives automatically, add the following lines:

```
30025 F1$="A:"+F$:F2$="B:"+F$
30035 SAVE F1$:SAVE F2$
```

These short examples are just two of the utilities you can write yourself to make your programming easier. This book should have given you ideas for others. The goal of the automatic IBM PC is to let the computer do all the work, and the programmer do all the creating.

Index

ISBN 0-8306-0898-2