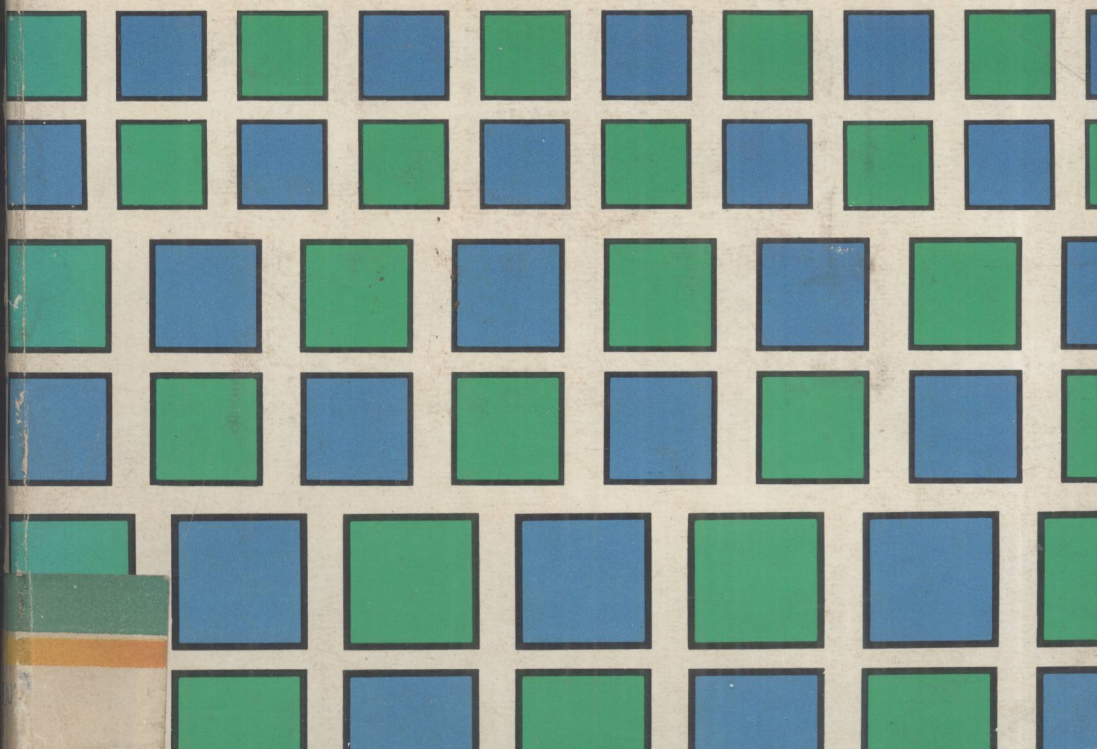
A REWARD BOOK

$7.

How to Debug Your Personal Computer

Jim Huffman/Robert C. Bruce

Do-it-yourself guide to debugging. Learn how to recognize, track, and eliminate problems in your program or system—the easy way!



How to Debug Your Personal Computer

Robert Bruce Jim Huffman

RESTON PUBLISHING, INC., Reston, Virginia
A Prentice-Hall Company

Library of Congress Cataloging in Publication Data

Bruce, Robert C
How to debug your personal computer.

Includes index.
1. Debugging in computer science.
I. Huffman, Jim. II. Title.
QA76.6. B775 001.64 80-18091
ISBN: 0-8359-2924-8

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia 22090

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

How to Debug Your Personal Computer

Introduction

A program that should but won't run on your computer is a source of building frustration, especially if you are just beginning to write your own or just paid good money for some new software! Short of starting again at Square One or sending the program back, what can you do about it? *Debug!*

This book will show you how you can tell when there is a bug in a program or your system, how to trakc it down to tis source, and how to get rid of it or get around it. If you have some knowledge of BASIC, you can put the techniques described in this book to work. All other information is included.

You would be absolutely correct in assuming that this is a book on the basics of debugging. In this respect, nothing is presumed. What we attempt to do here is show you how—step-by-step—to recognize and eliminate bugs when you're writing a program and how to spot them during the shakedown period. Also there are techniques for finding bugs in a program you didn't write, easy-to-follow guides to help you determine where in the program a bug is hiding and how to either eliminate it or work around it.

Of course, it is always easier to see bugs on a flowchart. In Chapter 1 you'll earn how to recognize them and how to construct a flowchart for and existing program. If the bug can't be eradicated by a change in the flowchart, the miost effective and accurate method of debugging is

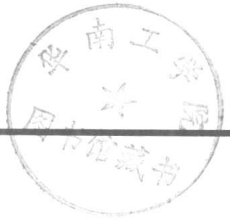
doing by hand with pencil and paper what the computer does electronically. You'll learn how to analyze a code sequence and follow the changing states of indexes and variables. Sometimes it is necessary to examine information stored inside the computer. Chapter 3 explains how to use print statements in an effort to track down even the most elusive bug.

And once in awhile, in spite of your best effort, a bug will successfully avoid correction. Most often you'll run into this situation in simulation and game programs where the program is more comples rather than a straightforwarrd series of events. In Chapter 4 you'll learn how to delvelop and integrate a patch to get around a problem that simply evades solution.

In Chapter 5 you'll learn where and how each debugging tool can be used most effectively. You'll go through a typical debugging effort and observe how a programmer uses the tools available to achieve an accurate cure for each situation in the shortest possible time.

At some point, you'll probably face a situation where a new or seldom-used program will not run on your computer. Where no bus appear obvious^ you may save frustrating hours by making sure your hardware is working properly. Chapter 6 explains the operation of the most popular peripheral units in an effort to lead you to a bug that is not in the programming. If you're a hobbyist or a beginning programmer, this book should serve you well as a basic handbook on debugging. You'll find more advanced techniques on debugging in *Software Debugging for Microcomputers* and *Personal Computing* (Reston Publishing Co.). The material in this book was drawn from these volumes.

Contents



1

Debugging the Flowchart

Debugging is not always performed after the fact; time spent catching potential bugs in the preliminary stage of program development could mean hours of time saved later on during the actual testing and shakedown stage.

The first step in writing a computer program is to list on paper all those tasks the program is expected to accomplish. This helps to organize thoughts and clarify objectives, and it gives the programmer a direction and a goal.

FLOWCHARTING THE EXISTING PROGRAM

Drawing up a flowchart first—before composing any lines of code—is a good way to catch potential errors in program flow, which should be welcome news since such bugs are usually the hardest to locate.

By the same token, generating a flowchart for an already written program is an excellent way to track down elusive bugs. The secret here is that since flowcharts are graphic representations of the flow of the program, bugs which can get lost in a jumble of statement numbers and

conditional branches become glaringly obvious once drawn in picture form.

Take as an example the following program. Even though it is well documented, it is long enough and complicated enough that unless we resort to graphing it out, we might never discover the bug.

```
10 REM TRIP CONTROL PROGRAM FOR
20 REM GREAT CENTRAL MODEL RAILROAD
30 PRINT "WELCOME ABOARD THE GREAT CENTRAL"
40 PRINT "MODEL RAILWAY. HOW MANY"
50 PRINT "ROUND TRIPS TODAY?"
60 INPUT T
70 REM T IS THE TRIP COUNTER
80 REM LOOP TO DETERMINE TRAIN LOCATION
90 REM DATA PORTS ARE
100 REM 1=IN STATION
110 REM 2=NEARING SW 1 FROM INNER LOOP
120 REM 3=NEARING TIGHT TURN
130 REM 4=NEARING STRAIGHTAWAY
140 REM 5=NEARING SW 2
150 REM 6=NEARING X-ING
160 REM 7=ON X-ING
170 REM 8=NEARING SW 1 FROM OUTER LOOP
180 REM 9=NEARING STATION
190 REM 10=THROTTLE
200 REM 11=X-ING SIGNAL
210 REM 12=SW1
220 REM 13=SW2
225 LET N=1
230 FOR I=1 TO T
240 LET N=-N
250 LET A=INP(1)
260 IF A=255 THEN GOSUB 440
270 LET B=INP(2)
280 IF B=255 THEN GOSUB 530
290 LET C=INP(3)
300 IF C=255 THEN GOSUB 660
310 LET D=INP(4)
320 IF D=255 THEN GOSUB 750
```

```
330 LET E=INP(5)
340 IF E=255 THEN GOSUB 840
350 LET F=INP(6)
360 IF F=255 THEN GOSUB 950
370 LET G=INP(8)
380 IF G=255 THEN GOSUB 1080
390 LET H=INPUT(9)
400 IF H=255 THEN GOTO 1120
410 REM POSITION SENSOR ACTIVATED PUTS
420 REM ALL ONES (255) ON DATA LINES
430 GOTO 250
434 NEXT I
436 GOTO 1280
440 PRINT ''ALL ABOARD''
450 REM WAIT FOR PASSENGERS TO BOARD
460 PAUSE 40
470 REM ACCELERATE OUT OF STATION TO SPEED 5
480 FOR J=1 TO 5
490 OUT 10, J
500 PAUSE 10
510 NEXT J
520 RETURN
530 REM NEARING SW 1
540 REM PULSE SW 1, ALL ONES
550 REM PULSES SW 1 TO INSIDE CIRCLE
560 OUT 12, 255
570 REM SLOW TRAIN ONE STEP
580 OUT 10, 4
590 REM HAVE WE CLEARED SENSOR 2 YET?
600 LET R=INP(2)
610 IF R>0 THEN GOTO 600
620 REM WAIT TO CLEAR SW 1 AND ACCEL.
630 PAUSE 10
640 OUT 10, 5
650 RETURN
660 REM NEARING RIGHT TURN
670 REM CUT TRAIN SPEED TO 2
680 FOR J=1 TO 3
690 LET S=INP(10)
```

```
700 LET S=S-1
710 OUT 10, S
720 PAUSE 10
730 NEXT J
740 RETURN
750 REM NEARING STRAIGHTAWAY
760 REM INCREASE SPEED TO 10
770 FOR J=1 TO 8
780 LET S=INP(10)
790 LET S=S+1
800 OUT 10, S
810 PAUSE 10
820 NEXT J
830 RETURN
840 REM NEARING SW 2
850 REM SET SW FOR OUTSIDE LOOP
860 IF T<0 THEN OUT 13, 0
870 REM CUT SPEED TO 5
880 FOR J=1 TO 5
890 LET S=INP(10)
900 LET S=S-1
910 OUT 10, S
920 PAUSE 10
930 NEXT J
940 RETURN
950 REM NEARING X-ING
960 REM SET WARNING FLASHER SWITCH
970 REM PORT 11 IS X-ING FLASHER
980 LET K=1
990 IF K>0 THEN GOTO 1020
1000 REM FLASHER ON
1010 OUT 11, 255
1020 PAUSE 5
1030 LET K=-K
1040 REM TEST IF WE HAVE CLEARED X-ING
1050 LET S=INP(7)
1060 IF S>0 THEN GOTO 990
1070 RETURN
1080 REM NEARING SW 1 FROM OUTSIDE
```

```
1090 REM SET SW 1 TO OUTSIDE
1100 OUT 12, 0
1110 RETURN
1120 REM NEARING STATION
1130 PRINT "NOW APPROACHING SMALLTOWN STATION"
1140 PRINT "SMALLTOWN, USA"
1150 REM SLOW TRAIN TO 1
1160 FOR J=1 TO 4
1170 LET S=INP(10)
1180 LET S=S-1
1190 OUT 10, S
1200 PAUSE 10
1210 NEXT J
1220 REM CHECK WHEN TRAIN MAKES STATION
1230 LET K=INP(1)
1240 IF K=0 THEN GOTO 1230
1250 REM STOP TRAIN
1260 OUT 10, 0
1270 GOTO 434
1280 PRINT "END OF TODAY'S RUN"
1290 END
```

At first glance, 129 lines of code may seem overpowering; but we will see that it only helps to point up the usefulness of flowcharting as a debugging tool.

Before we begin to generate our flowchart, we can learn a number of things about the program just by inspection. The program has been well documented, and we will use this documentation to our advantage as we go along.

We can tell, for instance, that the program seems to be divided into three distinct parts. The first part consists mainly of remarks explaining what the program is and what its assorted variables stand for. The second part is the main program (only about 15 lines long); and the third part consists of all those subroutines that were referenced in the main program.

Something which the program does not list, but which would be helpful to us as we try to visualize what is taking place, is a diagram of the track layout. The Great Central model railroad of the referenced program is shown schematically in Fig. 1-1.

The track has an inner route and an outer route. Both routes share a common side, the one which includes the tight turn and the straightaway. The inner loop passes by the station, where it must stop long enough for

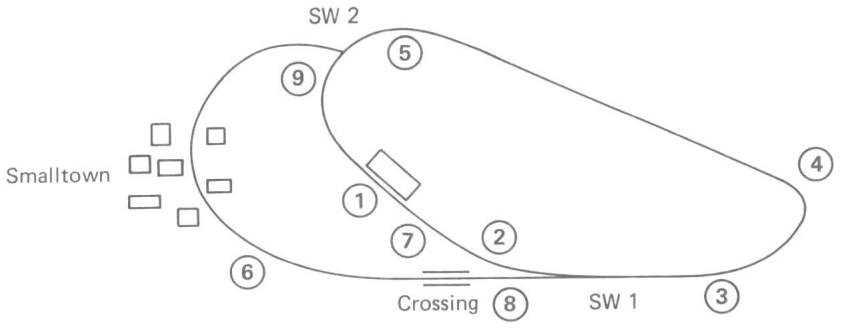


FIG. 1-1. Layout of the Great Central model railroad.

passengers to board and disembark. The outer loop carries the train through the manufacturing and business section of Smalltown.

The program was written to monitor and control the speed and location of the Great Central model railroad as it alternately traverses first the inner and then the outer loop of its right-of-way.

In order to monitor the train's location, sensors have been set up at various points along the track. Actuators, which respond to a signal of 0 (all zeros) or 255 (all ones) on their data lines, have been installed on both switches and on the railroad crossing warning flashers.

We may assume that all required analog-to-digital interfacing has been properly attended to, and as a result the only bugs (there are two) are attributable to software.

We begin with the first part of the program, the explanatory section. Flowcharting this (Fig. 1-2) is trivial, since the computer does not actually perform very much.

The second part of the program, the main part, is also not difficult to graph. Each sensor in turn is interrogated; and if the sensor transmits an all-ones activated signal (numerical value equal to 255), then the program branches to the appropriate subroutine before it continues the cycling sensor interrogation.

We can draw this portion of the flowchart as shown in Fig. 1-3.

DOCUMENTING

Since we were only trying to document the block of coding beginning at line 225 and ending at line 436, there were a few places that we were forced to leave blank. Specifically, we know that there is a NEXT statement to bracket the FOR statement:

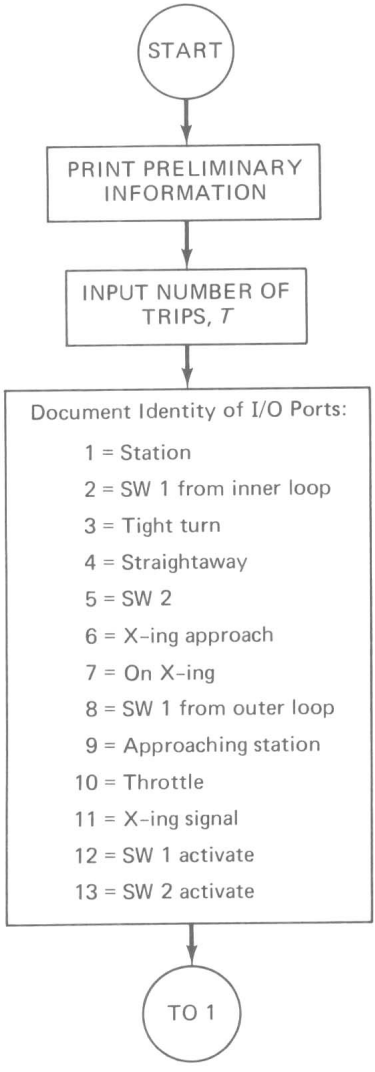


FIG. 1-2. Flowcharting the explanatory segment of the railroad program.

```
230 FOR I=1 TO T
    ≀
    ≀
434 NEXT I
```

But there is no statement in the block of coding we have just examined which sends the program down to line 434 so that the loop may be

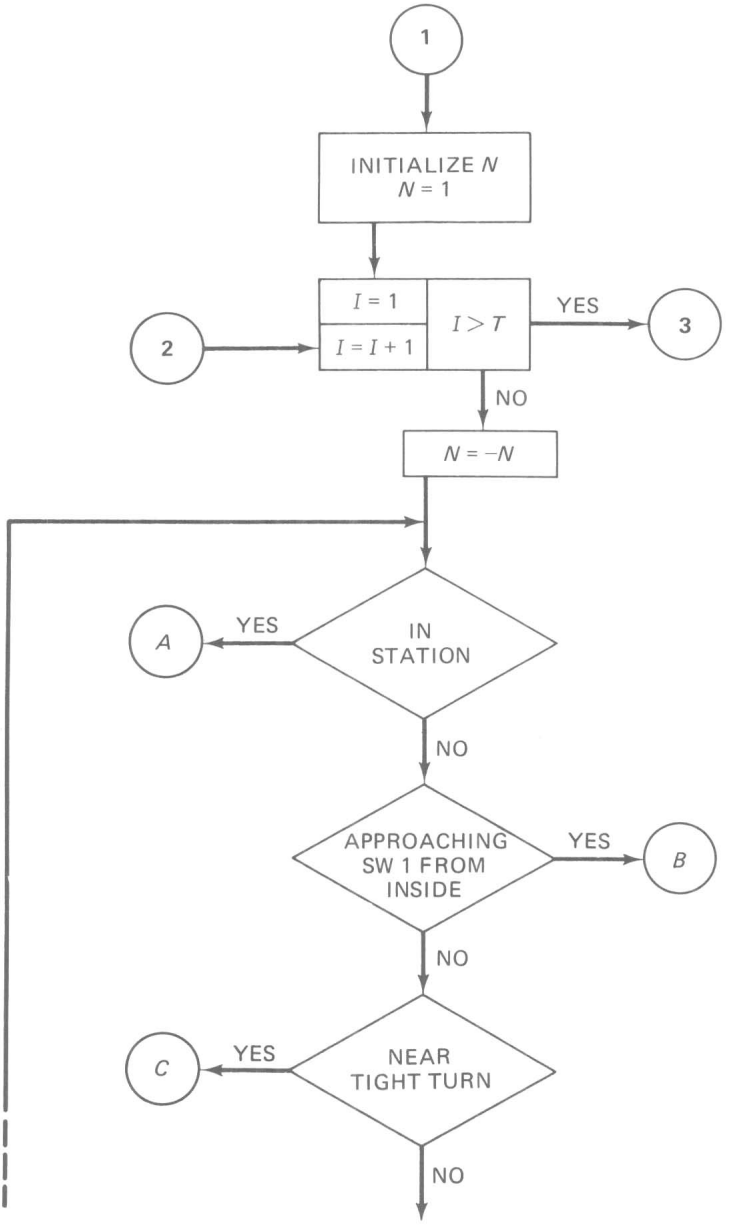


FIG. 1-3. Flowchart of the model railroad's main program.

incremented. We assume, then, that unless this is the bug, statement 434 must be entered from some other point in the program. We signify this by having a transfer symbol feed into the incrementing section of the loop symbol.

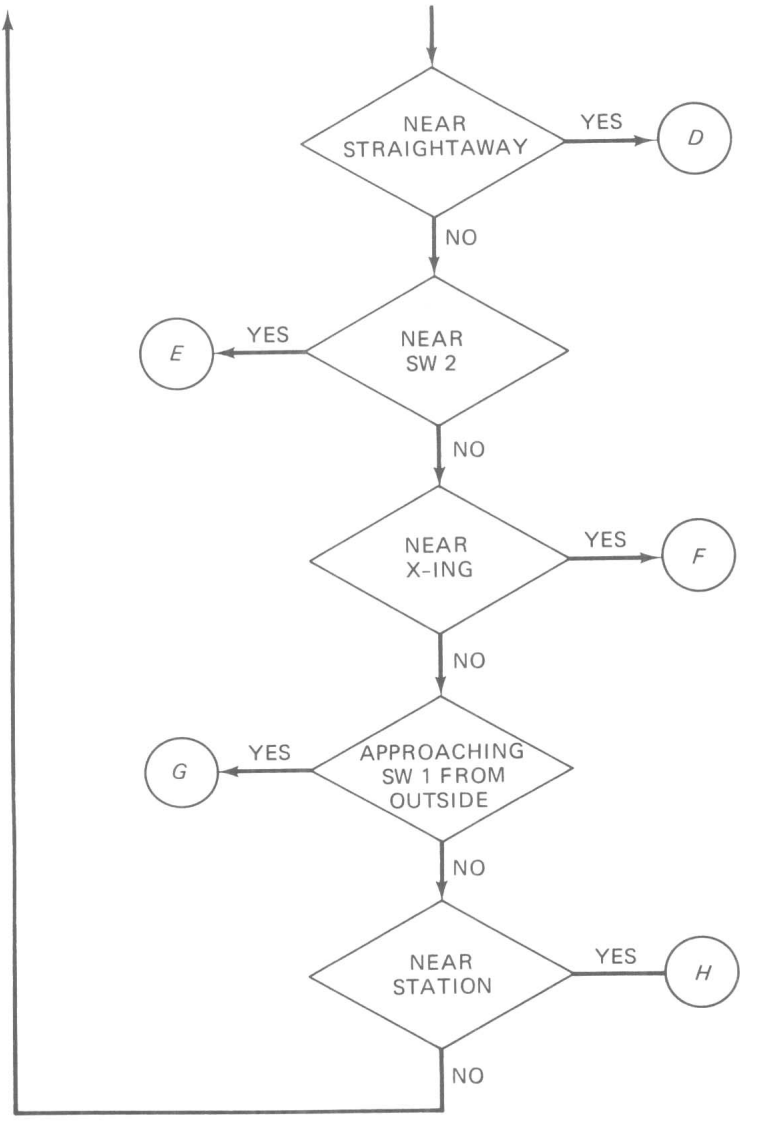


FIG. 1-3. (Cont'd.)

We note that the eight conditional branches themselves form a loop. For each IF statement, if the condition is met, program flow is transferred to a subroutine. If the first condition is not met, the next one is tried, and so on, apparently forever. This could be a bug: the program looks as though it is caught in an endless loop. Rather than jump to any conclusions, however, we should finish our flowcharting of the complete program.