1811

SECRETS OF SOFTWARE DEBUGGING

BY TRUCK SMITH

SECRETS OF SOFTWARE DEBUGGING

BY TRUCK SMITH

TAB | TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA 17214

Apple II and Applesoft are registered trademarks of Apple Computer Corporation.
IBM PC is a registered trademark of International Business Machines Corporation.
Z80 is a registered trademark of Zilog, Inc.
CP/M is a registered trademark of Digital Research Inc.
VIC 20 is a registered trademark of Commodore Business Machines.
Microsoft is a registered trademark of Microsoft Consumer Products.
UCSD Pascal and UCSD p-System are trademarks of the Regents of the University of California.
Visicalc is a registered trademark of VisiCorp.

FIRST EDITION

FIRST PRINTING

Copyright © 1984 by Truck Smith

Printed in the United States of America

Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.

Library of Congress Cataloging in Publication Data

Smith, Truck.

Secrets of software debugging.

Includes index.
1. Debugging in computer science. I. Title.
QA76.6.S6155 1984 001.64'2 84-8792
ISBN 0-8306-0811-7
ISBN 0-8306-1811-2 (pbk.)

Cover photograph by the Ziegler Photography Studio of Waynesboro, PA.

Acknowledgments

I would like to thank my sister and my wife. My sister made time in her extremely busy schedule to read and edit the book. She offered valuable comments on virtually every line of the manuscript. My wife managed her job, the house, and our new baby daughter—each one a full time job—to give me the time to write.

Introduction

This is a how-to book: how to debug. My purpose is to instill and increase debugging skills by analyzing debugging theory, discussing debugging practices, and showing many examples.

In our society, we have largely abandoned apprenticeship as an educational means, much to our detriment. An apprentice learns in two ways. He learns what the craftsman specifically teaches, and he learns through observation that which the craftsman finds difficult to explain. A great deal of information is lost when we try to teach only through books.

Debugging is a craft that is difficult to explain. In this book, I make liberal use of examples. The examples are in four popular computer languages: BASIC, Fortran, Pascal, and assembly language.

WHO IS THE BOOK FOR?

The book is intended for programmers at all levels of experience—novice, intermediate, and expert. The book only requires that the reader be reasonably comfortable with a computer language and rudimentary terminology. Basic concepts such as *branch statement* and *looping* are not defined.

In this book the novice will find comfort and guidance. The intermediate programmer will find a wealth of hints and ideas to make debugging easier. The expert will find an analysis and discussion of debugging skills not often presented, even informally.

ORGANIZATION

The six sections of the book are

Overview
Deductive thought
Knowledge
Debugging
Examples
Related topics:

The presentation proceeds from the general and theoretical to the specific and practical. The first section, "Overview," summarizes the debugging approach and highlights the necessary skills. In the

试读结束：需要全本请在线购买： www.ertongbook.com

next section, "Deductive Thought," I discuss the general problem solving skills that serve as underpinning for the specific problems of debugging.

Next comes the section entitled "Knowledge," which covers the computer knowledge that is helpful and sometimes necessary to the debugging task. The following section, "Debugging," discusses the specific tools and methods of debugging.

Because it is always easier to learn by observation and example than by reading theory, I've illustrated the complete debugging cycle using three sample programs: one in BASIC, one in Pascal, and one in assembly language. No matter what language you use, you can follow at least one discussion.

Finally the section entitled "Related Topics" discusses testing programs and proving programs. These concepts are put together in the last section to provide some additional insight into the process of creating reliable programs.

The book is structured so that each chapter builds on the one before it. This shouldn't discourage you from jumping around or turning to the chapters that interest you most; if you find that I'm using concepts that need further explaining, just backtrack or use the index to find where I explain them. Taken in order, however, the book presents a total theory of debugging and a practical approach to making your own programs work. The total theory and the approach should help your debugging more than the specific hints scattered throughout the book.

Most of the examples come from my own experience on computers from my Apple II, the office IBM PC, and Zilog Z80 to minicomputers and large mainframes built by IBM, SDS, Honeywell, Burroughs, and others. Both timesharing and batch environments are covered in the examples.

All of the examples are discussed in detail to enable you to understand them whether or not you are familiar with the computer or the computer language used. Most examples use BASIC, but some use Pascal and others use FORTRAN.

Welcome to the black art of debugging. You'll be challenged and frustrated, and you'll find fun and satisfaction with every step.

Contents

Acknowledgments vii

Introduction viii

Section I Overview 1

1 The Black Art of Software Debugging 3
An Example of Debugging—Debugging Is

2 Mental Attitude 8
The Big Lie—Believe in a Solution—Curiosity—Ego—Relaxation—Intuition

Section II Deductive Thought 15

3 Logic 17
Using Logic—What Is a Proof?—The Problem with Induction—Logical Analysis—ANDs, ORs, and Implications—Negative Inference—Pigeonholes—The Limits of Logic

4 Assumptions 24
It's the Computer—Identifying Your Assumptions—Unshakable Assumptions—When to Challenge an Assumption—It Really Is the Computer

5 Trial and Error 32
Guesses—Making Guesses—Testing Guesses—Getting Only One Answer—Controls

Section III Knowledge 37

6 Concepts 39
Layers—The Bottom Layer—The Top Layer—Data Representation

7 Attention to Detail 48
Punctuation—Variable Names—Similar Characters—Comments—Reserved Words—Logical Operators—Output—Reading the Manual

8 Program Structure 54
Grouping—Program Flow—The Effect of Structure on Errors

Section IV Debugging 67

9 Tools 69
Snapshots—Dynamic Tools—Interactive Tools

10 Methods 75
Describe the Problem—Guess Where the Problem Is—Guess What Might Be Going Wrong—Test Your Guesses—Refine and Repeat the Process—Determine the Fix—Weave It In

Section V Sample Programs 77

11 Comparing Files in BASIC 79
Program Design—The First BASIC Program—Debugging—The Second BASIC Program—More Debugging—Comprehensive Testing—Living Up to the Specifications—Summary

12 A Data Generation in Pascal 125
Backus Naur Form—Testing the Water—Program Design—The First Program—Syntax Errors—The Lights Go Out—Initial Debugging—The Revised Program: More Syntax Errors—Debugging—Summary

13 Radix Conversion in Assembly Language 196
The Program Design—The Pseudocode in BASIC—Debugging the BASIC Pseudocode—A Skeleton Assembly Language Program—Adding Single-Digit Multiplication

Section VI Related Topics 255

14 Program Testing 257
A Little Bit at a Time—All the Possibilities Every Time—All the Possible Paths—Border Points and End Points—Test Data

15 Proving Programs 260
The Stopping Problem—States—The Problem with GOTOs—A Sample Proof

Appendix A Pseudocode 264

Appendix B The Debugging Log 267

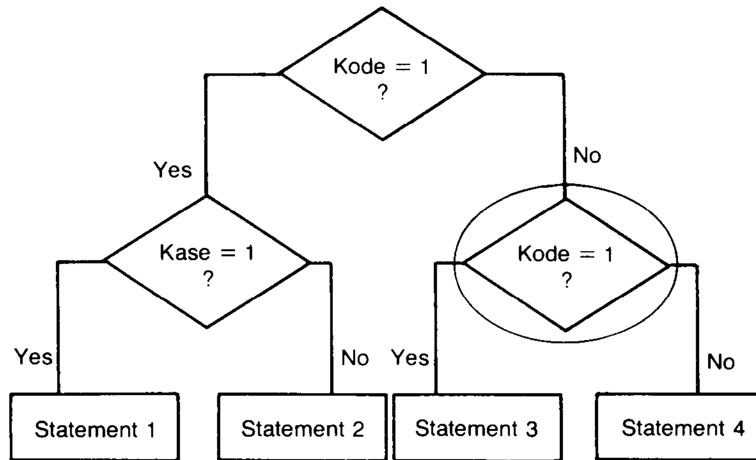
Appendix C Backus Naur Form 270

References 272

Index 274

Section I
Overview

Chapter 1



The Black Art of Software Debugging

It doesn't work.

You have just finished coding and typing in your program. Maybe it's 10 lines; maybe it's 10,000. Maybe you're working on your home microcomputer; maybe you're using the largest computer system in existence. The problem is the same; your program doesn't work.

Now what do you do?

Welcome to the black art of debugging. It's a black art because, although almost every programmer practices it, very few can tell you how to do it. Some people seem to have a knack for it, and others just flounder around changing things haphazardly until their program works. It's a black art in spite of the fact that debugging is really a very simple process. This book makes the process as painless as possible.

The basic job of the debugger is to identify the problem, find where in the program the problem is, and figure out what is going wrong so it can be fixed.

Debugging skill comes from ability in three areas:

- Logical thought
- Attention to detail
- Knowledge

In practice all debuggers use these three skills in greater or lesser amounts, and if you do any debugging, you do too. This book will help you organize your approach and show you how to use the three areas in your own debugging.

AN EXAMPLE OF DEBUGGING

To illustrate how the debugging process works, I'll take an example from my own experience. It's not a programming example, so it illustrates just how general the process of debugging is.

> My newborn daughter is crying. I have to find out why and do something about it.
>
> She's just been fed, so she's not crying because she's hungry.
>
> Are her diapers wet? No, and it's not because I've stuck her with a diaper pin either.

试读结束：需要全本请在线购买： www.ertongbook.com

Maybe she's cold. I'll wrap her in another blanket. She's still crying so that's not the problem.

Maybe she just wants to be held and rocked. That quiets her down for a minute, but then she's crying again.

Finally I try burping her. She stops crying and falls asleep.

In the process of solving my daughter's problem, I didn't just look at the major symptoms. I paid attention to the details, such as how long it had been since she'd been fed. I formed hypotheses about what was wrong (she's wet; she's cold) and tested them with logic. I used my knowledge: it would be hard to solve the problem of a crying baby unless you knew enough about babies to know which end the diaper goes on.

Because the problem itself was adequately (and loudly) defined by my daughter, I concentrated on discovering where the problem originated and what was wrong. I attacked the where and what questions simultaneously in a trial and error fashion, using what information I had or was able to observe to help direct my trial and error efforts.

This example illustrates the debugging process very well. Clearly debugging is both a logical process and an experimental process, and both parts are important.

The example also illustrates the importance of mental attitude. Just as in the example, you will find that most debugging is carried on under pressure. Usually you are up against a deadline (yours or someone else's). The situation is aggravated by the fact that the actual programming took longer than people expected it to. It is important to stay relaxed. Debugging can't be hurried, and a brute force approach may cause as many errors as it corrects. The best attitude for debugging is to be relaxed and rational.

DEBUGGING IS

Debugging is (choose all that apply):

- A black art
- A gift (some of us have it, others don't)
- A different process for each and every person
- Luck
- Natural ability
- Talent
- A skill that grows with time

It's easy to get mystical when you are confronted by someone who is good at debugging. You have to struggle and sweat to find one simple typo; the debugging expert flips through your program once and pinpoints the bug.

There are people who fill in the *New York Times* crossword puzzle in ink, too.

The truth is far more prosaic. Debugging is

- The ability to think logically; to puzzle out a problem
- The amount of knowledge you have (the more you bring to the problem, the better)
- Creative testing
- Attentive observation

Debugging is far more like solving a crossword or jigsaw puzzle than it is like sorcery. You do a lot of hard work and follow a lot of dead-end alleys until you stumble on the right one.

Is it possible to get a map? Is there any easy way to the solution?

Yes and no—there are certain ways that are easier than others, but you have to realize that you are going to have to do some work. You might as well go ahead and do it instead of vaguely tinkering with the program and hoping some brilliant solution will occur to you.

The debugging process follows these steps:

1. Describe the problem
2. Guess where it is
3. Guess what might be going wrong
4. Test your guess
5. Refine and repeat the process until you've found the error.
6. Determine the fix
7. Weave it in

These steps are not an algorithm even though

they look like one. They aren't an algorithm because the steps don't really happen one after another. They are not sequential; they sort of happen all at once, in an overlapping fashion. These steps aren't an algorithm because too much depends on the debugger making intuitive leaps in the process.

Why should that make any difference? Don't expect a plug and grind, simple formula answer to your debugging problems. Debugging requires exercise of creativity and intuition. This requirement is what makes debugging both fun and frustrating.

With that in mind, lets look at each step in more detail.

Describe the problem. Not many debuggers write down all the symptoms, but it isn't a bad idea. Number the incorrect results on your printout; write down the problems as they occur. Take note of all the ancillary circumstances too. For instance, if you are using a microcomputer, you may hear the disk drive start up. Sometimes such effects will give you clues concerning where the bug in your program may be.

Writing down a description forces you to look at and consider everything. Just as when you are doing a jigsaw puzzle, you need to look at both the shape of a piece and its color, when you are debugging you need to note everything you can about your program's misbehavior.

Get as complete a description of the problem as possible. Run as many test cases as you can; try to set limits on the incorrect behavior. See how many ways you can get your program to misbehave. Obtain lots of data to work with.

Of course you won't have too much to write down if your program dies completely after you type RUN, but you can write down everything you can about the way it dies. If you are on a timesharing system, sometimes the system will give you a location. If you are using a microcomputer, you can listen to see if the printer or the disk was accessed.

If you have only one run and one page of printout from it, check everything on the page for errors.

Guess where the problem is. This step and the next one both proceed simultaneously. You may know *where* to search for the problem but have no idea *what* caused the problem.

For instance, you may have print statements scattered throughout the program. If you get output from one print statement, but not from the one following, you have localized the problem. You still don't know what caused it.

One way of finding out the location of a problem is to deliberately place print statements throughout the program. Another way is to do a trace on selected variables.

Sometimes you have a guess about *what* causes the problem that gives you an idea *where* to search. For example, you may identify an incorrect answer on your printout and say to yourself, "That's calculated in the blank routine. I'll look there."

No matter what the order is; you have to know where the problem is before you can fix it.

Guess what might be going wrong. As I mentioned above, this step and the one above usually proceed in synch. Sometimes you take one look at your program's results, and you know just exactly what the matter is. That in turn identifies where the problem is. Other times you can tell where the problem is, but you don't know what causes it. If you know where the problem is, you can make a guess about what's going wrong, based on what you know about the problem.

Whether you think you understand the nature of the problem or you merely know its location, you should make a guess about the problem. Not too many debuggers write down their guesses. They think, "It must be in that group of routines," or "It looks like this variable is not getting set correctly." Sometimes the guess cannot even be coherently expressed; the guess may be a vague hunch about some section of the program.

Writing down your guess is not vital. But writing it down will sharpen your awareness of the nature of your guess and lay the guess open to more careful analysis.

These are preliminary guesses. Eventually as you repeat the process and refine your guesses, you reach the kind of guess that goes, "The problem is in line 4320," or "The variable KDC was used in the subroutine call, not KDS." From the beginning of

the guess-making process to its end, there is a continual narrowing in on the right answer, the problem solution.

Test your guess. Once you have a guess, you have to test it. You may check over your program listing. You may make a change to your program or run it with a trace. You don't necessarily have to change the program to test your guess. Sometimes the guess will imply things about your last run of the program that you can check on the printout.

You may have noticed that some variables were being printed incorrectly. Your guess may imply that there should be certain values for those variables, perhaps equal to another variable or perhaps powers of two; there are any number of possibilities. You can go back to the printout and check. If you don't have a printout, go back to the problem description you wrote down in step 1.

Toward the end of your debugging session your guess will be specific enough that the test will mean inserting a fix. Most often your test will mean inserting a print statement somewhere or using a trace to see if the program flow holds true. You will test many guesses in the course of solving a difficult problem. Devise tests that will give you more information about your problem and point you more decisively toward the solution.

Refine and repeat. The refine and repeat step is a constant narrowing in on the problem. It won't always seem so. Sometimes a guess will lead you down a wrong path. Sometimes you will spend hour after hour with no apparent progress. Then, all of a sudden, you will have a flash of inspiration that makes everything clear.

The false starts and hours spent going nowhere are not wasted time. They are part of the process. Sometimes they provide the necessary information to make the final brilliant insight possible. Sometimes they just convince you that there really is a problem to be corrected. Either way, some progress is being made.

Whenever it is possible, your guesses and tests should be designed to carry you further toward the solution—to make it easier for you to refine your guesses and tests.

A straightforward example of this test design is dictionary search. If I am looking for a word in the dictionary, it will not help to let the dictionary repeatedly fall open to a random page and see if my word is there. In practice most of us use a successive refinement approach. If our word is in the last half of the alphabet, we open the dictionary toward the end. We then use the information gained from that page to specify where we turn next; if we are far from our word, we will turn more pages than we would if we were near it.

When you are debugging, you continue to refine your guesses until you have identified the portion of the program responsible for the error. This portion might be as small as one character, or it might be a whole routine that wasn't completely thought out when it was originally written. Either way, you stop guessing once the problem has been identified.

Determine the fix. If your problem was a simple one, you may have determined the solution in the process of identifying where and what the problem is. A typo for instance is easily corrected once spotted.

The fix is not always so simple. Perhaps you have neglected to take care of some special cases in your algorithm, or maybe the algorithm only works for easy cases. Sooner or later you will have a bug that will demand that you completely rewrite a routine.

Take care in tackling such cases. It is not safe to blindly insert a few lines into an existing routine; nor is it ever safe to completely rewrite a routine under the pressure of a debugging deadline. You have to decide which alternative to choose; whether to insert a few lines or rewrite. Granted, some of the factors influencing your choice will be psychological, but take as much time as you need to analyze your fix. As a rule of thumb spend as much time on the fix as you spend on your original algorithm.

Weave it in. Weaving in the fix is just an extension of determining the fix, but it is important enough to rate a step all its own. Don't blindly insert a correction. *Understand* the code you are trying to correct; make the correction a reweaving job not a patch.

The extreme example of doing this badly is the programmer who discovers that a variable is being set to the wrong value. He doesn't know why. He doesn't care why. To solve the problem, he simply inserts a line that sets the variable to the right value.

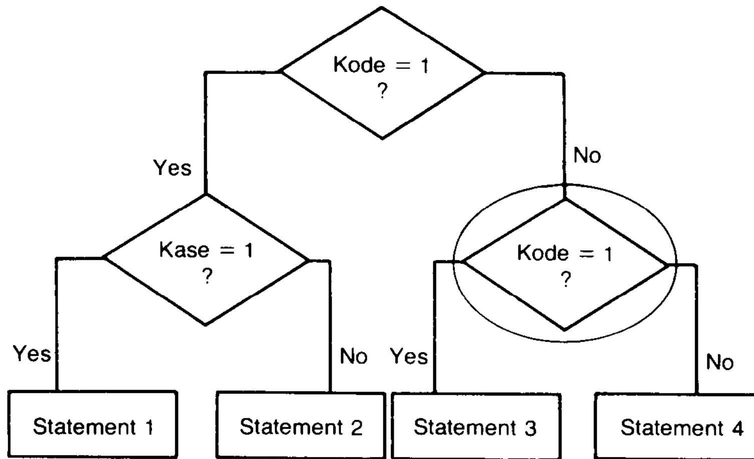
This practice is not only a barbaric programming and debugging technique; it also creates problems later. Suppose another error crops up (and it will; there is always one more error no matter how clean the program is). Our hapless debugger (or the poor person who has to follow in his footsteps) now has to figure out what is wrong in code that isn't structurally sound, and the "correction" may disguise the new error's source entirely. Even if the two errors are unrelated, the correction makes the problem more difficult to solve because the debugger is never sure that the two errors aren't caused by the same portion of code.

Even if no further errors crop up, the same problems occur if someone wants to add to or change the program.

The solution? Understand your program and weave your corrections in so that they become an integral part of the program. If you are debugging someone else's program, weaving in corrections means using the same programming style as the original programmer used. Although this may seem to carry the weaving idea to extremes, it's not a bad extreme to go to; you make the corrected program easier to read and debug because you haven't inserted any jarring notes.

No matter how you debug, how experienced a programmer you are, or what machines or languages you program, you will follow a process like the one outlined above. This common sense overview describes the backbone of the debugging process. The rest of this book elaborates on elements of the process in order to give you the skills that will make the whole process faster and easier.

Chapter 2



Mental Attitude

There is nothing more important to debugging than your mental attitude. All the technical skill in the world will not help, unless you are willing to approach your debugging problems in a rational and relaxed fashion.

In this chapter I will talk about what it means to be rational and relaxed, and how to maintain a rational and relaxed attitude.

THE BIG LIE

In the world of human affairs there are two approaches to any problem. The first is the scientific approach; you assume that the problem has a solution that is accessible to all. You attempt to find that solution. The second approach is political. Knowing that the truth is fuzzy, you essentially stand up and shout your position until everyone agrees with you. This second approach is the big lie.

I'm not saying that all politicians use the big lie or that all scientists are completely logical. There is a distinction, however, between the two different types of mental attitudes and approaches to problems.

No one person consistently uses one viewpoint or the other. The attitude of any given person will vary from moment to moment depending on the problem, the emotional situation, and other factors.

What attitude is best when you are addressing the computer? Clearly not the political one; no amount of smiling or kissing babies will get results. For example, no matter how many times you retype the line

```
L=LEN(S$
```

to a BASIC interpreter, it will give you an error. You can shout, plead with the machine, or kick it. You can convince everyone else that you're right, but the computer will remain stubborn until you put the final parenthesis on the line:

```
L=LEN(S$)
```

Only then will the computer accept the line.

This example is not meant to indicate that dealing with the computer is all black and white. There is usually more than one way to solve a problem. Sometimes the computer will make mistakes. Sometimes the system itself is wrong.

In my office there is a ZILOG Z80 based system. Whenever we turn it on, it executes a self test program. Three times in ten, the self test program says there is something wrong with the computer.

Our standard response is to restart the system. On the second try the system usually works.

It's not a question of whether or not there is something wrong with the system—there probably is (we'll just have to wait until it gets bad enough to interfere with operations before we get service—but by telling the computer it's wrong (restarting it), we have chosen a political solution.

In practice you will find the political approach is often tried. This shouldn't necessarily be thought a waste of time; it's often necessary to try the political approach just to convince yourself that there really is a problem..

For example I have a Pascal system for my Apple II. In my initial experimenting with the system, I took one of the programs that comes with the system and tried to modify it. All went well until I tried to save the modified program. The system gave me an error message:

```
FILE ERROR - CAN'T SAVE PUSH
<SPACEBAR> TO CONTINUE
```

I tried to save the program again—and again, and again. It took three tries to convince myself that the system wasn't just kidding me. I am reasonably logical and levelheaded. But, because I didn't have any immediate solutions to the problem—didn't, in fact, have a glimmer of an idea why I got that error—I procrastinated and kept trying the same "solution" until I convinced myself I would have to find another way.

In this instance the political approach cost me some time, but it also convinced me that I would have to find a solution. Some people would be convinced on the first try, some on the third; some will never be convinced no matter how many times they try. This last type of person has the most problems in debugging.

Stick to the scientific approach. Problems with the computer and your program are most often amenable to reason. Consider the problem a puzzle to be solved, not a situation in which the computer can be convinced to behave by repetition, shouting, or force.

Conversely, if you don't immediately try the scientific rational approach, don't be too hard on yourself. Tell yourself (as I have and do) that you are dumb, but don't take it too seriously. Sometimes it is necessary to make mistakes to find the right answer. The reasons may be psychological; we may need to convince ourself that there truly is something wrong. The reasons may be accidental; we may need to try several approaches before we stumble onto one that works. Mistakes are part of the debugging process. Be tolerant of errors and don't be overwhelmed by them.

BELIEVE IN A SOLUTION

Once you're convinced you have a problem, you have to convince yourself that a solution exists. Believing in a solution has two parts; one, that there is a solution after all, and two, that it's not some trick that someone's played on you.

A Solution Does Exist

It is difficult to believe that a solution does exist when it's 2 A.M. and you've been searching for it since 8 the previous morning. To remind yourself that a solution exists, remember the most extreme possibility; you can rewrite the whole offending routine. I don't recommend rewriting the whole routine (studies show that it's a good way to introduce more errors than you started with) but it is an alternative.

It's especially hard to believe that a solution exists when you think it might be a system error—something wrong with the computer or a limitation imposed by the computer. Beware of laying the blame elsewhere so you can avoid responsibility. When you're confronted with what seems to be a system error, try and prove that it is.

Take as an example one of the times I sus-

试读结束：需要全本请在线购买： www.ertongbook.com