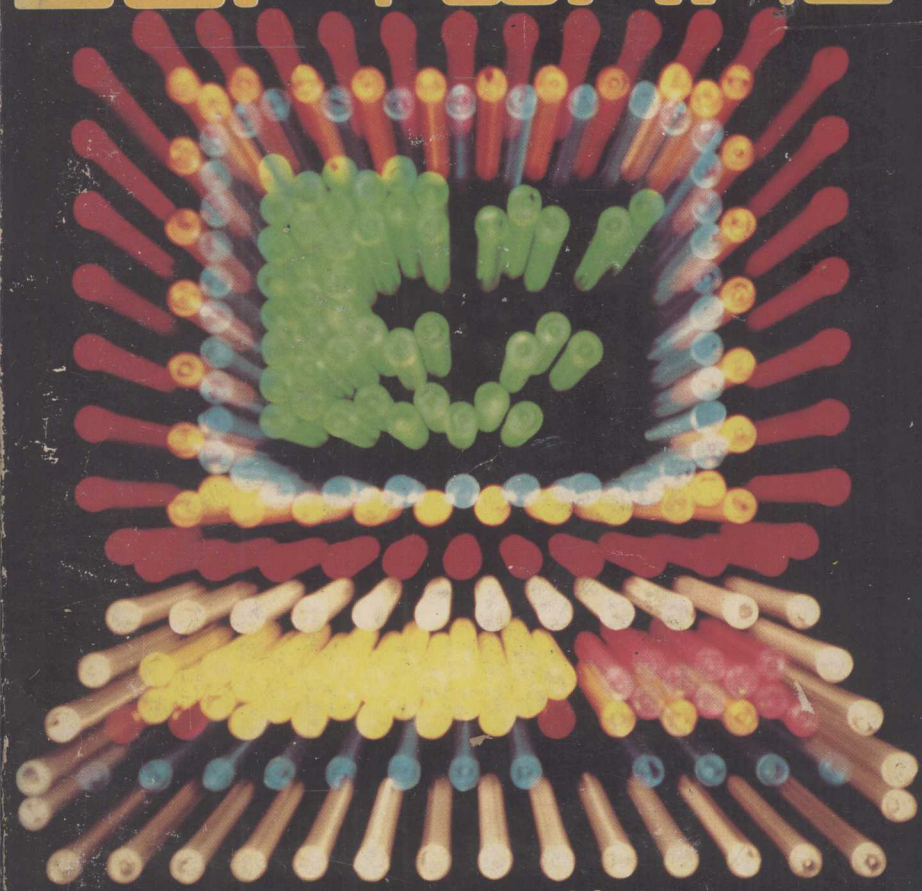
TAB 1369

$13.95

THE GIANT BOOK OF COMPUTER SOFTWARE



Step-by-step guide to creating *your own* computer programs!

EDITORS OF 73 MAGAZINE

THE GIANT BOOK OF COMPUTER SOFTWARE

THE GIANT BOOK OF COMPUTER SOFTWARE

BY EDITORS OF 73 MAGAZINE

TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

THIRD PRINTING

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Main entry under title:

The Giant book of computer software.

"TAB book #1369."
Includes index.
1. Electronic digital computers—Programming.
I. 73 magazine for radio amateurs.
QA76.6.G5 001.64'2 80-28843
ISBN 0-8306-9627-X
ISBN 0-8306-1369-2 (pbk.)

Contents

Preface

Now that you own a microcomputer—or even just a programmable hand calculator—what are you going to do with it? Do you know how to program it? Chances are that you're a tenderfoot with software—the programs. If computers themselves have lost their mystery, software somehow still manages to baffle most computerists or prospective computerists.

This book will take you by your hand through programming. You'll find plenty of examples to guide you. And every example has a blow-by-blow description of what the computer program is doing in each program line. This is an excellent way to learn about software while having some fun, too.

No matter what you're other interests are, the programs in this book offer you a little bit of everything. And don't forget that with a little work here and there, almost any program can be modified for a different situation from the one for which it was originally written. Good luck!

Chapter 1
Introduction To Computer Languages

Anyone who wants to use a computer has to have a way to communicate with it. This is a simple introduction to some of the languages which are used for that purpose. It is intended for rank beginners, so all of the programmers, software freaks and computer hot dogs in the audience might as well stop here. For anyone else, I'm going to try to keep everything in English (which is not a computer language, unfortunately) and avoid as much computerese as possible. So here goes.

One might start by asking, "Why have computer languages at all?" Back in the dark ages 25 or 30 years ago they didn't—the machines were wired up to do a certain thing and that's what they did. But, somewhere along the road, some bright fellow realized that it would be much more efficient if you could feed the machine a fairly long set of instructions and let it follow them. This also made for much greater flexibility, because you could give the machine different sets of instructions. These instructions are what a computer language communicates, and this section will cover some of the more common ones, to wit: Assembler, BASIC, FORTRAN, PL/1, COBOL and a little bit about some of the more specialized ones. But first let's look a little bit more at the nature of the beast we're dealing with.

FIRST, THE COMPUTER

It is helpful to think of a computer as a glorified electronic calculator. In fact, some of the more modern calculators really are

computers. But let's look at the average four-function calculator. It has a display, which is called an output device in computerese, and it has a keyboard, which is an input device. To do anything with it, you have to enter the numbers through the keyboard and then enter what you want done with them, be it to add them or whatever.

But suppose you want to do a mortgage calculation—say, figure out what your interest payments and principal payments are going to be each month for the life of the 20-year mortgage. This means that you're going to have to do a very repetitive calculation 240 times. It is to avoid this sort of hassle that real computers (and some fancy calculators) have *stored program capacity*. A stored program is just a set of instructions inside the machine which get done without your standing there pushing all of the buttons each time. But now we have to get this set of instructions inside the machine, and that is where *programming languages* come in. We need a way to communicate the instructions to the machine.

MACHINE LANGUAGE

In the case of a calculator, this isn't much of a problem. The *instruction set* (list of all of the instructions which the machine is able to follow) is hard-wired in. If you want it to add, you push +. This is obviously not too good an idea for a large computer, because the number of buttons gets pretty large. Besides, you tend to run out of symbols, which makes everything even more confusing. So we need some sort of language. The simplest one is called *machine language* and is the closest we can get to what the machine actually speaks. However, computers and other digital machines run on high and low levels of voltage. This is rather hard to see, so we usually represent it with ones and zeros, or on and off lights. The whole thing is like trying to communicate using RTTY (radioteletype) but doing it by ear instead of with a machine, which is to say that it's a royal pain. Anyone who's into interpreting things like 0100110101100001 and so forth can really get off on it, but for most of us there's gotta be a better way. Fortunately there is. Incidentally, those switches and lights on the front panel of any early microcomputer are used to communicate with the thing in machine language.

ASSEMBLY LANGUAGE

The next sort of language developed, and the one which is most widely available for microprocessors these days, is the *assembler*. Each type of computer has its own version; what it is, in

short, is machine level logic—but using symbols and normal numbers rather than ones and zilches. Assembler is related to what you do with a calculator in fact, any of you who own or use HP calculators have been using a version of assembler language usually known as *reverse Polish notation*. With an assembler language, you specify what number you want, where you want it put and what you want done with it; for example (to use HP assembler): "12, ENTER (which puts it in the region where the arithmetic is done), 2, ×" multiplies 12 times 2 and comes up with 24. All assembler languages work this way, although many of them have dozens of commands and hundreds of locations where things can be put or obtained. This sort of computer language has a lot of advantages. It's very efficient not only where memory is concerned, but also with regard to execution time. The assembler (the program which translates it into machine language) doesn't take up much memory either, which means it can be used in a microprocessor that doesn't have much memory (and memory costs like the devil, even these days). Using assembler, you can also anticipate situations where the machine might do something unexpected, since you're on the machine's logical level. Of course, it's got its problems too. It's hard to learn, not easy to use well, hard to debug (find errors) and is "machine dependent," which means that each machine has its own. To sum it up, a lot of people don't like to have to write: "12, ENTER, 2, ×." They'd rather write "A =12×2." This is what *high-level languages* let you do, along with all sorts of other convenient things. For this reason, almost all programming these days is done with one of the various high-level languages, and the section will cover some of the more common ones.

HIGH-LEVEL LANGUAGES

First of all, a high-level language is a computer language that is based on some combination of English and algebra. So, to write two plus two you would usually write "2+2." To tell the machine to print, you write PRINT, WRITE or something similar. The one thing to watch out for is that, although there are many different ways of writing one thing in English (and to a certain extent in algebra), a high-level computer language has a very narrowly defined structure and vocabulary. This means that the computer equivalent of "I ain't got none" will be rejected. In other words, you have to be very careful when writing any sort of program for a computer, because errors (the computerese term is *glitches*) get caught faster than they would be by an old-fashioned high school English teacher.

Continuing the comparison with human languages, there are lots of different ones for computers, too. At first each company developed its own; now many of them are standard and thus can be used on any machine with few, if any, changes—unlike assembler languages. We still have a lot of computer languages, though. For example, the last time I checked the documentation, there were something like 35 different high-level languages available for use with the University of Michigan computer system. The reason for having so many is that each language is designed to do some particular thing well (in jargon, they are problem-based rather than machine-based). This means that one language is good for mathematics (also called number-crunching), one is good for electronic circuit design, another for library use, and so forth. There are also a couple of general-purpose languages. These happen to be the most popular for obvious reasons.

It would be a pain to have to spend a few weeks mastering a new computer language every time you wanted to do something different with a computer, not to mention that you have to buy (or write) a special translating program (called a *compiler*) for each one, and that can get very expensive (much more than the cost of the computer itself). So let's look at some useful, fairly general purpose languages.

BASIC

BASIC, which stands for *Beginners' All-purpose Symbolic Instruction Code,* was developed at Dartmouth College in 1965. It was designed for people who knew nothing about computers but who wanted to use them. Few dyed-in-the-wool programmers care much for BASIC, but most non-programmers love it. It's based on algebra and the non-algebra parts of it are in plain English. For example, to enter two numbers into the machine, multiply them, divide one by the other and print the results, you would write:

```
1 IN A, B
2 LET C=A*B
3 LET D=A/B
4 PRINT C,D
```

(* is the standard symbol for "times" on a computer)

This language has quite a few advantages. It's easy to learn, easy to use, and there are lots of books around which help people learn it. Equally important, there are lots of programs already written and published in it (these are called *canned programs*), and

all microcomputers can use it. The compiler (remember, that's the program which translates the things you write into the machine's language) doesn't take up too much memory either, which means that BASIC is suitable for small computer systems where memory is limited. A final advantage is that BASIC is fairly flexible, especially the advanced systems. You can do many—if not most—of the same things with it as you could do with FORTRAN or PL/1, although the programming effort might be greater.

It does have some disadvantages, though. BASIC has no mnemonic variables. This means that you have to remember that A stands for current, E for voltage and so on. In a more advanced language you could write AMPS, EMF, etc. This isn't so bad if you're working with standardized symbols, but gets to be a disadvantage when you try to remember which was accounts payable and which was accounts receivable. Also, BASIC is somewhat limited as to what you can do with input/output. This only makes a difference if you are working with a big system that gives you lots of choices; it isn't of too much concern for a home computer system or a small business one. Finally, BASIC is structured somewhat along the same lines as FORTRAN, which is the oldest computer language still in use. This means that is does a lot of things in more difficult and more roundabout ways than some of the newer languages, like PL/1. For example, its "either-or" choice is rather cumbersome to write. It's still a great language to play around with, though.

FORTRAN

FORTRAN is probably the best known of the various computer languages, partly because it's one of the oldest. The name stands for FORmula TRANslation, and it was developed by IBM back in the early 1950s. It and the B-Zero language developed by Univac were the first high-level languages used. No one uses B-Zero today (few have even heard of it), but FORTRAN is probably the most widely used computer language in the U.S. Of course, the FORTRAN we use now isn't the same as the FORTRAN introduced back in 1957—just as the English we speak now isn't the same language as the people in England spoke back in 1066. There have been three official versions of FORTRAN: FORTRAN (the original), FORTRAN II and FORTRAN IV. Number three got lost in the middle somewhere. Most computers these days use FORTRAN IV.

Anyway, as you might guess from the name of the beast, FORTRAN is basically a scientific computer language; it was

developed to make it easier to solve mathematical-type problems for science and engineering. Over the years the language has expanded to the point where it is usable as a general-purpose language, so it can do a lot more than simply crunch numbers. Moreover, because it's such a popular language, there are who-knows-how-many programs written (and sometimes published) in it, which makes it much easier to solve a given problem (since you can frequently just type in a canned program). The same structural problems encountered in BASIC are part of FORTRAN, but these have already been covered. Perhaps more important for anyone who wanted to use FORTRAN on a microcomputer, the compiler (remember that's the program which translates it into machine language) takes up much more memory than a BASIC compiler, though much less than one for most other high-level languages.

While FORTRAN was developed for scientific use, COBOL was developed for business use. It's the language generally used by the U.S. government, so it's got a pretty wide circulation. Needless to say, many businesses use it, too. From the little work I've done with it myself, it seems that you spend most of your time defining what your printout is going to look like and what the information which you feed into the thing is going to look like (the jargon for this is *format definition*). It also takes lots more memory than one would probably want to pay for in a microcomputer, unless he wanted to do subcontracting for the government.

Up until the early 1960s, most computers were either scientific- or business-oriented, and a machine which was designed for one didn't usually work too well for the other. Then IBM started making their general-purpose machines, and most other people followed suit. At about the same time, people started to worry about a general-purpose computer language. A number were developed, of which my favorite (just for the name) is MAD (Michigan Algorithmic Decoder) which was brought to us by the folks at the University of Michigan Computing Center. Then IBM got into the act, and, lo and behold, out popped PL/1 (Programming Language One). The best description that I can think of is that PL/1 was designed to out-fortran FORTRAN and to out-cobol COBOL all at the same time. It does a pretty good job of it, too. I'm always amazed at all of the nice things you can do with PL/1; to use the computerese phrase, it has more bells and whistles (extra options) than you can shake a stick at. Unfortunately, it also uses more memory than you can shake a stick at, which makes it normally too

expensive for microcomputer use (or even time sharing use if you have to watch your costs).

To give an example of the sort of nice thing the language can do (among others) it lets you write a simple either/or statement (if this is true, do one thing; otherwise do this other thing), whereas to do that in most other languages you have to play hopscotch with the line numbers. In short, PL/1 is a great language.

Another new language, again by IBM, is APL. I will confess here and now that I've never used it, so what I say is taken from what people who have used it have told me. This is a very powerful language; it can do in one line what most other languages require five or more to do. It is not yet widely used.

I mentioned a while back that, in addition to the general-purpose languages I've discussed so far, there are quite a few special-purpose ones. For a hobby user (or would-be user) these aren't too important, but just to be more or less complete I'll mention a few that are good with which to impress people (besides being good to know about if you tend to be around computer hot dogs who like to talk about such things). There is RPG, which is a Report Generating language, and therefore much used for business and that sort of computing. Then there are several languages used to write *simulations*. A simulation is like a computer game except people take it seriously. These languages are things like GASP, GPSS and so forth. There are a fair number of languages used for crunching words instead of numbers: SNOBOL, SPITBOL and so forth.

As a brief review, one needs some sort of language to give a computer instructions. You can operate on the machine's level (called machine language) and feed it ones and zeros. Or, you can stay on the machine's logical level but use decimal numbers and abbreviated commands. This is called an assembler language, and it always requires a separate program for translation into machine language. Finally, you can use a combination of human logic and normal words and symbols, which is called a high-level language. This requires a program to translate it into machine language again, and such a program is called a compiler. The principal advantages of high-level languages are that they are easy to learn, easy to use, and are the same for any machine. They are not nearly so efficient as assembler language from the computer's point of view, but they are much more efficient from our point of view—and that's usually what counts.

MEANINGFUL CONVERSATIONS WITH YOUR COMPUTER

The computer freak now has a wide choice of micro kits, input/output devices, and services to choose from, but the big problem remains: How does one COMMUNICATE with his computer? Ah, yes, you have just brought home your new microcomputer and plan to use it to store data. A simple application? Yes, but there your micro sits, not calculating a thing. Something is missing—a method of "talking" to, or *programming* the machine, thus enabling it to perform a meaningful task.

The Language Processor

The missing link is the *language processor* (LP). An LP is a program that allows a computer user to form the unique set of machine instructions, which are the binary numbers that direct every machine function. The LP is to a computer what the keyboard is to your pocket calculator; it is a man-machine interface that bridges the gap between human requests and machine action. Without a keyboard, your pocket wonder is useless unless you like to collect big IC chips. Without an LP, a computer programmer is forced to form and insert into memory every binary code that forms a program, requiring a knowledge of the internal construction and machine codes unique to every computer.

Programming Your Black Box

A computer performs its task by executing a series of machine instructions that reside in the memory of the processor. The ultimate goal of any programmer is to relate the problem to be solved in terms of the machine instructions that the computer understands. This goal may be reached in two ways: by inserting each instruction into memory by hand, using the front panel switches, after a tedious process of forming the correct codes; or by using an LP to form the codes for you. This LP may be an *assembler, compiler,* or *interpreter*. This article examines, in simple terms, the function of each type of LP, the associated trade-offs, and the benefits of each. Before starting, however, let's take a quick look at how programming works without the LP, bearing in mind that the final goal of any LP is to produce the binary machine code that only the machine understands.

Machine Language

Every computer, be it an IBM 370 or Motorola M6800 micro, has a unique set of instructions, consisting of binary codes that