# RHUG

# GAZETTE

The Official Newsletter of the
Richmond Heath User's Group

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

Volume I Issue 3
February 1982

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


Good day and welcome to the next installment of the GAZETTE. We're glad you could make it for our wonderful journey through the world of sizzling electrons and their glorious trainers. Speaking of making it (quit the snickering, kids)...

## Meeting Notice!!

The next meeting of RHUG will be on FEBRUARY 8th at the illustrious Alpha Audio studios in scenic midtown Richmond, VEE-AYE. For those who don't know where it is, try 2049 West Broad St. If you're not familiar with it, it's right down the block from Fifth Avenue (where most of us will be both before AND after, I'm sure). It will be at 7:00 as usual. Ready to find out how you get in? Catch this one...on the side of the door there is a touchpad. If you carefully punch in 4302 (the night's secret code) the door will unlock magically. This system is compliments of Carlos and Bobby who have nothing better to do than sit around and figure out better door locks. They have used the same touchpad set-up for a better mousetrap but found that the mice could not be trained to remember the access codes so they sold the idea to D-Con who is going to market it next year for their Roach Motel (the exclusive version) since they discovered that roaches CAN remember these things. The meeting will be on the third floor (turn left at the top of the stairs) in the new conference room. If you're not careful they will record the meeting and blackmail us all. REMEMBER: It was decided at the last meeting that anyone who does not show up is considered fair game and may be talked about as much as the participants want with no legal recourse left to the non-attending member.

* * * * * * * * * * * * * * * * * * *

## Bytes On A Budget...

### How Much Is That Software In The Window?

The rhyme or reason to the price of personal computer software has puzzled the majority of computer owners I know for sometime now. It seems that there are two or three second generation text editors and three or four professional word processors and scads of game disks (or cassettes) which range in cost from free to a telephone number length price tag that offer NO real performance guarantees. We see ads like, "BETTER THAN VISICALC!", "SUPERCEDES THE INDUSTRY STANDARD!", and "NOW VERSION 3.0!" which makes you ask: Does it cost three times as much or is it the first version that really worked?. Or finally, how much are they going to hit you up for version 4.0 if you buy this one

At first I thought the length of time it took the programmer to author and code the program might be a key to these costs, but after seeing some of the adventure type games I've checked out, i.e., Remarkable Experience, I realized that this cannot be the case. Remarkable Experience is a large game and it utilizes a data file requiring over 3K of memory just for pointer information! The binary file and data base eat up over half of a 5" disk of room and, get this,>>>>$15<<< is the going rate. On the other hand, I recently saw a Blackjack for an Apple 2 which cost $23.95 and was labeled "auto deck shuffling". I thought we wrote that in the Heath Basic course...

I could go on but I think I've made my point. YOU CANNOT JUDGE SOFTWARE BY IT'S PRICE! (Ed. Note: Chafin's 1st Law of Software) It also seems that some of the least talked about software is some of the best. Some of this is due to a lack of marketing prowess on the part of small vendors. Some is overlooked in the flood of product offered in the trades and newsletters where it is publicized. And some is not considered because the purchaser is afraid of support from small and remote vendors even if the product is superior to similar programs. The reason I am touching on this right now is that I feel I am beginning to understand the overall picture a little better (I hope?) so for starters herein lies what I believe to be the first of three possible causes for the small system pricing phenomenon:

One-Certain programmers are nice guys and just wnt to share their programs with others and be helpful. (Applause!)
Two-Certain programmers are optimistic and/or greedy and charge too much, or the programmer is making very little and a third party marketing firm is raking off too much cream!
Three-The programmer and his software marketing organization research the market, price the software fairly, and the end-user receives a working product which gives that user a fair return on his software investment.

The third condition is a logical extension of a maturing and expanding industry and I feel that we will see more sophisticated and more expensive software offerings. My only hope is that with more elaborate software beginning to dominate more and more space in our newsletters and magazines my first conclusion will not fade away leaving the hardware tinkerer and the beginning computerite to only find **It is now pitch black, if you proceed you are likely to fall in a pit!**

Happy 'putering,
Carlos

* * * * * * * * * * * * * * * * * * *

## THAT'S (NOT QUITE) INCREDIBLE!

### Words of Wisdom from Bobby Tulloh

Well, here I sit, staring blankly at the screen, surrounded by all these sophisticated electronic devices, trying to write the next installment in this series. (Voice from Ether: "Series? I've never seen this column before! What is this anyway?") Quiet! OK, this is the first installment of T(N/Q)I [Ed. note: What does that formula DO, Bobby?], and in it I intend to discuss (and hopefully help clear up) some problems/peculiarities of Heath/Zenith CP/M, and pass along some tricks or hints that might help assembly language programmers. I will pass along any little goodies that anyone wishes to share (giving proper credit, of course, and a gold asterisk) [Ed: How do we accomplish THAT one?] and welcome feedback or input.

Since this column is supposed to be a source of info and not just an outlet for an over-active character generator [Ed: You said it...] let's talk about the new Bios we all received recently. The main reasons for the .03 Bios are to support the soft-sectored controller card for the H89/90 and the hard disk. Now those of us with H8's say "What's that do for me?". Well, not much except give you interrupt driven terminal I/O, a software clock, an event counter and the ability to cause a warm boot using the break key. Not bad, huh? Well we better look at all this one piece at a time.

The Good News: Heath/Zenith is apparantly going to come out with a soft-sectored controller board for the H8, contrary to the original asnnouncement that the H8 would not be supported after the introduction of the H100 Supercomputer. (Source: H-SCOOP). This disk controller is rumored to be a high priority item at Benton Harbor.

More Good News: Ralph Sampson has unofficially announced that he will play basketball at UVA until he graduates in 1983. If this is true I think the Cavaliers could beat the Boston Celtics next year. I feel this could be the beginning of a dynasty. Dean, Lefty, Denny, et al---Eat your hearts out! Wahoo Wa!

[Ed: Did I miss something here?...]

Sorry! Got side-tracked. Can you tell I'm a basketball fan? Can you spell NCAA?

Back to the discussion at hand. (That's the trouble with these word processing syystems - They take over after a while and start creating all by themselves. Who is winning the information revolution? Do you know? Who's side are you on?)

Before I interrupted myself, we were talking about the new Bios. The four new features are not extremely visible or highly documented. I don't know why, because I feel that all of them are very useful. That is, they would be useful if they were actually installed. Let me clarify what I mean by this. I have seen three different distribution versions of this Bios and each one has been different (Is that clear now?!). One supported the event counter (more on that next month), one supported interrupt input and the event counter, and the last one (mine) did nothing. I didn't find that exactly thrilling. (I am also using Tandon double-sided drives whihch .03 only supports for the H37 soft sector card, so all I got from the update was a few new disks--not really...)

Interrupt-driven keyborad input - Quite handy. For HDOS folks I don't need to explain why, but for CP/M'ers: This allows you to enter data or responses more quickly, in many cases without waiting for prompts. This is nice in programs where you

give familiar sequences or responses. It allows you to enter commands or data while the system is doing other tasks (usually) so you don't have to wait for prompts.

Time-of-Day Clock - So far unsupported but easy enough to use. This is based on the TICCNI routines used for system timing and is fairly accurate. It is, I'm sure, going to be used for a future (although available now from other sources) hardware clock/calendar, but can be used by certain application programs as it stands.

Warm Boot w/Break Key - This is a great idea! This, along with interrupt keyborad input, will get you out of the infinite loop without a reset and cold boot. I know a few times when I've needed that!

Do not attempt to try any of this at home! If these options are available in your version, the sign-on messaage will tell you. Otherwise FORGET IT!! Next month I will explain how to implement these things on your system and possible reasons why you shouldn't, plus talk more about the event counter. I will discuss adding the new options to the old Bios for people who are using Tandon drives.

Now, to close, just a small programming proverb from the Rosetta Stone:

"NEVER WRITE A SUBROUTINE THAT WON'T ALLOW ZERO ITERATIONS"

* * * * * * * * * * * * * * * * * * * * * *

## What Makes a Good Program

By Jim Scott

### INTRODUCTION

This is the second of a series of articles on the principles of programming. The first article, "How to Write a Program", outlined the process of writing a program from conception to debugging. This month's article will consider some of the goals that a programmer should pursue during the programming process.

But first, let me tie up a few loose ends from last month.

### AFTERTHOUGHTS ON THE PREVIOUS ARTICLE

I neglected to mention in the first article that this series will not attempt to teach the specifics of a particular programming language. That is a very different topic and could be the basis for a future series of articles. For now, I will try to make sure that any examples I use are self-explanatory.

Also, a couple of additions to last month's article seem necessary. I would like to add two more stages to my outline of the process of writing a program. Here is the revised outline, with the two new entries marked with asterisks:

-Recognizing a need
-Thinking about the problem
*Developing program specifications
-Designing the program
-Choosing a language
-Coding
-Keying
*Eyeballing
-Debugging

The process of "developing program specifications" should take place before the designing of the program begins. Program specifications are a set of written details about the input and output of the program and the processing required to create the output.

For example, if the program is to play a space war game, the program specifications should provide answers to questions such as these:

-What are the exact rules of the game?

-Will the program play against a human opponent, or will it provide for two humans to play against each other?
-What should the output look like? (e.g., will it be a picture of a grid representing a quadrant of space or a sequence of sentences describing what is happening?
-Will graphics or color be used?
-Exactly how does a player enter a move?

And so on. The specifications should say everything there is to say about what the program is to do but not necessarily anything about the methods to be used. Or to put it still another way, the specifications tell how the program is to behave as far as a user is concerned, as if it were a black box.

Developing program specifications and the stage I call "Thinking about the problem" usually overlap quite a lot; they tend to take place at the same time. Nevertheless, to emphasize that they are two quite separate activities, at least conceptually, I will refer to them as different stages.

The other new stage, "Eyeballing", should take place between keying and debugging although many programmers make the mistake of skipping it. It consists of a careful study of the source program after it has been completely keyed in but before any attempt to compile or run it.

(The "source program" is what we call the result of the coding and keying stages. This is to distinguish it from the "object program" or "executable program", for example an .ABS or .COM file. In the case of a typical Basic program, there is no object program; the source program is executed or interpreted directly. "Coding" is the process of writing a source program and "code" is a source program or part of one.)

Eyeballing is also commonly known as "desk checking". This term dates from before microcomputers; back then, if you were checking your program at your desk you obviously weren't using the computer.

In most cases, eyeballing will greatly shorten the time it takes to get a "clean" source program, i.e., one with no syntax errors. It may also catch some of the logic errors that would otherwise have to be rooted out during the debugging stage.

(Syntax errors are analogous to grammatical errors in English. Similarly, logic errors are analogous to incorrect meaning in English. Either may be called a bug.)

Eyeballing can be done either from a printout of the source program, or at the terminal (e.g., by using the text-editor to view a screenful at a time). In either case, it should be done slowly and carefully; it should not consist of just a quick scan. And it should involve the entire source program, not just parts of it.

Look for misspelled variable names, missing punctuation, and in general, any kind of mistakes. The types of syntax errors you should watch for will depend on the programming language. Review the program logic at key points, and assure yourself that they will do what you want them to do.

It will also be tempting to skip the eyeballing stage, but to do so is false economy.

Eyeballing becomes especially important when you consider the software that is generally available for home computers. An interpreter (for example, any Basic interpreter) does not always scan the entire program; it starts at the beginning and runs until an instruction tells it to stop, or until it hits an error. Therefore, you generally catch only one bug with each attempt to run the program. Many of the compilers available also stop at the first syntax error. This, incidentally, is a sign of a cheaply-written compiler; quality compilers, like all assemblers I have encountered, continue to process the program in spite of syntax errors, allowing you to correct many errors for each attempt to compile the program. (I will define the differences among interpreters, compilers and assemblers in next month's article, "Language Differences".)

Now, on to this month's topic...

## What Makes a Program Good

I will list and explain several attributes which I believe any program should have if it is to be considered a "good" program. The frightening thing is that there may not be any programs anywhere that have all these attributes. Certainly the great majority of programs violate the heck out of several of these attributes. You may be asking yourself why you, especially if you are a novice, should try to follow guidelines ignored by many professional programmers. The answer lies in the fact that each of these attributes is intended, directly or indirectly, to prevent program errors. If you have any concept of the problems caused every day by program errors throughout the world, you will want to clutch this article to your

breast and maybe even do some of what it says.

The list is somewhat arbitrary, because there is a lot of overlap and because there are different ways of expressing the same ideas. Here is my list of the attributes of a good program:

A good program works.
- does what it's supposed to do.
- does what it's supposed to do all the time.
- does what it's supposed to do in all reasonable environments.
- is usable.
- is documented.
- is structured.
- is modular.
- is readable.
- is maintainable.

Now here is what I mean by each of these attributes.

## A Good Program Works

Obviously a program should work. But this is a matter of degree. Let's start with the lowest level. A program that starts by displaying garbage on the screen, then crashes the system, is not real useful. Yet there are certainly programs in existence which absolutely do not work; mostly, they are written by would-be programmers who think that coding and keying are the only stages in the programming process.

## It Does What It's Supposed To Do

A program should do what the program specifications say it should do. That's obvious too, right? But this criterion eliminates a large percentage of all commercially available programs. Another way of saying it is that the program design stage should follow the program specifications and the user's manual. Most large programs vary from the written specifications just enough to be aggravating; some fall so far short of doing what they are supposed to do that selling them might very well qualify as fraud.

## It Does What It's Supposed To Do All The Time

Not only should the program follow the program specifications, but it should follow them all the time. It should handle all kinds of input, including erroneous input. In other words, it should not have bugs. Unfortunately this criterion eliminates most of the programs ever written. Probably the only really bug-free programs are trivial ones.

## It Does What It's Supposed To Do In All Reasonable Environments

The program specifications should say which type of hardware and operating system the program will work with. Will it run on H8's as well as H89's? What about TRS-80's? How much memory will it require? Will it run under CP/M? HDOS? What versions of each? Will it use cassette or disk? What kind of disk? How many drives? Which terminals will be supported? Will it make any difference whether the user has the Extended Configuration Option? Or the Trionyx mother board? Or dynamic memory?

It is up to you (or your boss or client) to decide which hardware and software environments to support. If the program is just for one user then it only has to support that user's environment. If you are going to advertise and sell the program you may want to support a wide range of environments; but don't claim support for environments you haven't tested it in.

Support for multiple environments implies that, during the design and coding stages, you avoid the use of techniques that are dependent on your particular hardware or operating system. In particular, the use of absolute memory addresses (for example, the use of PEEK or POKE in Basic) greatly restricts the range of systems on which a program can be run.

## It Is Usable

The program specifications should be written with consideration for human factors. That is, the program should take account of the fact that its user is a human. A human is slow (compared to a computer), but impatient when the program is slow. A human makes mistakes, but expects the program to deal gracefully with them (and not to make any of its own).

Therefore, the program should minimize response time (the amount of time elapsed between the moment the user presses "Return" and when the output becomes available). When mechanical operation is involved, such as copying a disk, response time of several minutes may be acceptable. But when the processing involves only the computer itself (central processing unit) and memory, response time of more than two or three seconds becomes excessive.

The program should be prepared to handle any kind of input the user can dish out. If the program has displayed the question "Want to play again?" perhaps the only valid answers are "Y" and "N". But it would be better to accept other reasonable answers like "YES" and "NO" or even "OK". In any case, if an invalid answer is typed, such as "HUH" or "3" or control-Z or Return (no answer), the following are acceptable, human-oriented responses:

-Repeat the question and ask for a proper reply.
-Explain why the answer typed was wrong, then ask again.
-Make a safe assumption (e.g., that the user wants to play again).

The following are NOT human-oriented responses to invalid user input:

-Crash the system.
-Terminate the program.
-Display a sarcastic error message (Ed: That's half the fun...)
-Make a rash assumption (e.g., in a text-editor program, assuming that all the updates typed in for the last three hours are to be discarded).

There are many more aspects to program usability. You can probably think of many examples of programs that are unnecessarily difficult to use. For example:

-The operating system that requires several replies every time you boot it.
-The dump utility that displays the contents of a file in hex but not the ASCII (character) equivalents.
-The compiler that quits at the first syntax error.
-The text editor that requires memorization of a large set of single- aracter commands.
-The game that takes 20 minutes to calculate the program's move.
-The assembler that prints a cross-reference, but not in alphabetical order.

Until you have enough experience to take account of human factors in your program specifications, be prepared to spend some time trying out your debugged program, then making changes to make it more usable.

## It Is Documented

Every program should have user documentation and programmer documentation. User documentation is a written manual or instruction sheet which tells the purpose of the program and how to use it. To a certain extent, user documentation can be included in the program itself. For example, complete instructions can be displayed at the user's option when the program starts running; or the program can be menu-driven. But there should always be a separate document that allows the potentiasl user to decide whether the program is of use to him or her, and describes at least in general terms how the program is used.

Programmer documentation describes the structure and logic of the program so that the original programmer or others can maintain it (i.e., fix bugs or make modifications). At least some of the programmer documentation will be in the source program itself, in the form of blocks of comments (or "remarks") as well as comments on individual program statements. If these comments are sufficiently complete, perhaps no programmer documentation other than the source program itself is needed. In fact, there is an advantage to making the source program self-sufficient in this respect--anybody who has the source program will have all the information needed to maintain it.

Of course, if the source program is not distributed, e.g. if you sell the .ABS or .COM file only, the user will need only the user documentation, not the programmer documentation. But the programmer documentation must exist, because someone will have to maintain the program.

## It Is Structured

The term "structured programming" refers to a set of guidelines for writing code using straightforward, easy-to-follow logic. Depending on the programming language, structured programming is often closely associated with avoidance of the GOTO statement. Structured programming is the topic of a future article.

## It Is Modular

Modular programming is another set of coding guidelines. A modular program is divided into logical units, or modules, with each module being a maximum of about a page of code or less. Subroutines and functions are two types of modules. Again, this is the topic of a future article.

## It Is Readable

The source program should be coded in such a way that the eye of the reader is guided to the divisions between logically separate pieces of the program. Use of blank lines, indentation, and boxes around blocks of comments are very valuable techniques. The use of structured programming and modular programming are essential to readability.

In addition to the visual segmentation of the program, the use of comments and mnemonic variable names helps to make it understandable. (A "mnemonic" variable name is one which suggests the purpose of the variable. For example, if a numerical variable will contain a checkbook balance, "BALANCE" is a more mnemonic name than "B" or C123".)

## It Is Maintainable

As mentioned above, "maintenance" refers to two areas of activity that take place after the program has been "finished" and put into use. The first is the ongoing fixing of bugs that inevitably turn up. The second is modification or enhancement of the program, to provide new functions or to allow it to run in new environments. In many cases, the maintenance of a program occupies far more programming time in the long run than the creation of the original "finished" product. ("CP/M", considered as a program, is an excellent example. It has been reworked many times since its original development to adapt it to run on many types of microcomputers what didn't even exist when it was first designed.)

So it is very important that a program be designed and coded in such a way that the effort required to maintain it is minimized. This is much the same as saying that it should be readable, either to the original programmer after several weeks, or to a programmer who has never seen the program before. If a program works properly and is documented, structured, modular and readable it probably is inherently maintainable.

(Coming Soon: "Language Differences")

* * * * * * * * * * * * * * * * * * * *

## The Editor's Coroner

This time around (and maybe for ever and ever) this column's name has been changed. As long as I feel like lukewarm death it's an a propos title. I was planning on elucidating you with words of wisdom and incredible insights into the wonderful and wacky world of computers but this newsletter has gone on long enough. I simply refuse to add to it. You will have to wait. Something to look forward to. You RHuger's will have to hang in for another month. Then maybe we will have an Editor's Coroner which will amaze and astound you and even fool your friends.

In the meantime, we wait with baited breath for your comments, input, correspondance and scatalogical missives. Send all of them to:

Richmond Heath User's Group
c/o Jim Scott
1724 Blakemore Road
Richmond, Virginia 23225

## LATER, FOLKS!!!

(c)1982 Richmond Heath User's Group
Contents may only be reproduced with proper credit to both the individual author and the Richmond Heath User's Group but are not to be reproduced for commercial use without the express permission of RHug.