The Open Channel is exactly what the name implies; a forum for the free exchange of technical ideas. Try to hold your contributions to one page maximum in the final magazine format (about 1000 words). If you want to include illustrations, we'll accept anything (short of libel or obscenity) as long as it's submitted by a member of the Computer Society. If it's really bizarre we may require you to get another member to cosponsor your item.

Send everything to Jim Haynes, Applied Sciences, UC Santa Cruz, CA 95064.

Dear IEEE:

Readers of your magazine can obtain free programs by sending a copy of a program they wrote. They will receive a copy of someone else's program that was sent in and their program will be sent on.

Readers must include a large stamped manila self-addressed envelope, or else a big nothing will be sent! Acceptable forms are paper tape ASCII, typewritten, or printout. Prefer Basic but any language or system O.K. Send to Soft Swap Shop, P.O. Box 4834, Arcata, CA 95521.

Now we know what to do with those reels of 7 track paper tape we had left over when we got rid of the Burroughs 220. J.H.

A Tale of Two Computers

which we shall identify as Brand X and Brand Y, probably without concealing much of anything.

I have a program I run from time to time, written in PL/1 for Brand X. I have never bothered to rewrite it for Brand Y, because Y doesn't have PL/1, and I don't use it that often anyway. Besides, it has always worked before on the Brand X machine. Typical running time is 20 minutes.

The last time I ran it, the computer operator came up to me and inquired discreetly if it might be in a loop, since it had been running for 45 minutes. We agreed he should kill it. I took the output, such as it was, to the systems programmer and asked him just where my program was executing when it got into the loop. I might mention in passing that the Brand Y machine would have been able to tell me where the program was executing even before it was killed, because the Y system keeps the segment number and the program counter relative to the beginning of the segment at run time, so that one can go right to the compiler listing. But Brand X doesn't work that way.

I might also mention my initial surprise that my program had run for 45 minutes at all, because I had put a 30-minute time limit on the JOB card; but then I remembered that the time limit field hasn't worked for about the last five software releases, so now people are supposed to put the time limit in a separate TIME = field on the card. But my program had never got itself into a loop before, so I had never bothered to change the JOB card, and anyway I didn't know offhand where to look up the rules for where to put the TIME = field on the job card.

Well, the system programmer told me he couldn't tell from my meager output just what was going on, but that I should run the job again with an ABDUMP card in the deck, and that would cause it to output enough information so that he could probably pinpoint the loop. I had to wait until the next day before our shop could accommodate another 45-minute run. (Brand Y is a multiprogramming machine; but Brand X would need a lot more core than we have to run more than one program at a time, so we have to reserve most of the day for short-turnaround jobs.) I told the operator to let it run for 45 minutes again and then kill it. I was surprised that there wasn't more output, having seen dumps before, but again I carried what there was to the system programmer's office.

"This isn't any more than you had yesterday," he said. "I can't tell anything from this; it's just the register contents from the operating system after your job had already been flushed out."

"But I put in the ABDUMP card, just like you said." "Oh, I see what's wrong. Besides the ABDUMP card you also have to say 'MSGLEVEL = 1,' instead of 'MSGLEVEL = 1' on your EXEC card, or you don't get the dump. I forgot to tell you that."

I had plenty of other things to keep me busy while I waited another day for a 45-minute shot at the machine. This time the job didn't run at all. Something wrong with the job control cards. Back to the system programmer.

"Oh, yeah. You can say 'MSGLEVEL = 1,' but when you want to say 1, 1 you have to put it in parentheses, like 'MSGLEVEL = (1, 1).' "

Unabridged source code & documentation for 8080
Basic published

The complete documentation plus a complete, annotated, 46-page assembler listing of a Basic interpreter for the 8080 has been published in Dr. Dobbs Journal of Computer Calisthenics & Orthodontia. According to Dr. Dobbs Editor Jim Warren, this particular Basic interpreter was developed in 1976 by the Lawrence Livermore Laboratory—in particular, John Dickenson, Jerry Barber, John Teeter, and Eugene Fisher.

The Basic interpreter is designed to operate on an MCS-8080 system. It is "pure code"; that is, it may be placed in ROM or PROM. It requires five kilobytes of storage for the interpreter. The interpreter includes a complete floating point package. The documentation and listing of the floating point routines are included in the materials published in Dr. Dobbs's Journal... which is published ten times a year by People's Computer Company.

The release didn't say which issue carried the Basic interpreter, but I suspect Jim Warren will be happy to tell you if you write him at PCC, Box E, Menlo Park, CA 94025. J.H.
I was able to fix the card and get back on the machine without losing another day. At last I was rewarded with quite a lot of output after the operator killed the job. Now I was sure the system programmer could help me.

"This will take a while," he said. "Could you come back tomorrow?"

Next morning I was standing at his office door when he arrived. "Oh, I couldn't do anything with your problem last night. You gave me the listing and the dump output, but I need the load map, too."

"Golly, I don't have that anymore. It's been a year since I compiled the program; and anyway those things are just gibberish to me, so I never save them if the program seems to run all right when I try it. And so I had to compile and link-edit the program again to get a load map. Fortunately I had long ago broken the program up into separately compiled modules, so this one took only 10 minutes to compile and load. I laid the output at the feet of the system programmer and told him I would return the next day."

On the morrow he handed me my listing, with a few lines circled in red. "It's probably somewhere in there," said he.

I thanked him and went back to my office, rather proud of myself for having written a program this well structured, with not a single GOTO. The efforts were well rewarded. That same afternoon I saw the place where the program might be running off the end of an array, as a result of reading an input value that I hadn't planned to use when I wrote the program, but which might occur now because of some changes in the application. I increased the array size, and also added a statement to check the value of the array index against the bound before using it to address the array value. Later, someone was to tell me that I could declare something about SUBSCRIPT-RANGE and the PL/I compiler would have included code to check for the out-of-bounds possibility; but I had never happened across that topic myself in reading the PL/1 manual. Indeed, I confess to never having read the manual cover to cover, since so little of what it says seems to have anything to do with the program I am trying to write. Let it be noted in passing that the Brand Y machine checks for invalid index values automatically, in the hardware, and would have killed my program immediately if that were really the cause of the problem. So I recompiled, and link-edited the new version into my library. I assured the operator that this time it would run to completion in half-an-hour or less.

It's fortunate that I was delayed in leaving for lunch, because the operator came to me just as I was leaving.

"It's run for almost 45 minutes again. What should I do with it this time?" he asked.

"You might as well kill it," I replied, not feeling so confident as I had an hour before. I took the compiler listing to lunch with me. I couldn't imagine why my repair to the program had not fixed the problem.

I was rather red-eyed the next morning, and beginning to lose my enthusiasm for the Thrill of the Chase after the elusive program bug. I just couldn't see any way the program could go wrong, outside of the subscript problem that I had already fixed; and my hand-executions with various combinations of correct and incorrect data had done anything unexpected. I decided it would be necessary to make a test run with statements added to the program to print out various values that might give me some clue to the nature of the problem.

This raised a tactical question: should I aim only to print values when something might be wrong, or should I print out values for every input item? I knew the latter course would produce far too much data for me to examine, because the program had run successfully many times with thousands of input items. On the other hand, my trap for unexpected index values had failed to work even when the program had run for 45 minutes; so maybe I was looking in entirely the wrong place. For this reason I opted in favor of more output. I set up the job as a compile-and-go. This time it quite properly killed itself after a minute or two of running, having produced as much output as the output parameter on the JOBO card would allow. Everything looked perfectly reasonable. If the program was going into a loop it was apparently happening only after running correctly for quite some time.

I gave some thought to getting rid of most of the input data and running with only the items near the end of the input, in hopes of catching the troublesome item with less running time. That turned out not to be very practical in this instance, because the input data to this step is produced by a couple of SORT/MERGE operations interspersed with other programs. I couldn't see any way to hack up the input data so as to produce only the last part of the input to the program in trouble. I suppose there is a utility program somewhere that would let me copy the file after the other programs have produced it, throwing away the first part of it in the process; but I couldn't make any sense out of the utilities manual, and nobody I asked had ever done such a thing either. I even thought of writing the data out to tape and then taking the scissors to the tape: but the operators convinced me that I would never be able to read the tape after cutting away the part containing the labels.
After lots and lots of hard thinking about how to get the program to tell me what was going wrong without wasting a box of paper (not to mention the time I would have to spend reading through a box of paper), I hit upon a plan. I would put a test in the program that would stop it and print out helpful values if the critical subscript value exceeded the original array dimension, before I had increased the size of the array. This got results! The program ran for about 25 minutes and stopped itself, printing out the expected values. Sure enough, the subscript value was two or three places beyond the end of the array. I hadn’t thought to print out the value that was actually obtained from the array using this subscript value, but I could readily believe it would be such as to cause the program to loop indefinitely.

This leaves us with just one mystery: why didn’t increasing the array size fix the program? I had already run it with a test for the subscript value exceeding the new, larger array bound, and it had never reported anything wrong. Could it be that (a) I had fixed one bug, but another one elsewhere was causing the program to loop? or (b) perhaps the program was not in a loop, but I had grossly underestimated the running time with the present set of input data? The former seemed likely enough, except that the system programmer had told me it was running right in the area where I found the bug at the time when it was killed. This seemed to indicate that I was searching in the right vicinity. The latter possibility didn’t seem at all likely; it had never run much longer than 20 minutes before, the input data collection was not much larger than usual this time, and it had been near the end that one time that I had successfully detected the invalid index value.

There was one more thing to try. I had not gone back to the system programmer with a dump when the program was killed after I had corrected it. I couldn’t locate that particular listing and load map in the great pile of them that had accumulated by now, but one more compile would provide them. I scheduled another 45-minute run. By now I guess the system programmer was ready to get rid of me and my problem, because he went right to work on it as soon as I brought in the dump. The spot he circled in red this time was near the one he had marked before, although they were not quite the same. What could possibly be going on?

I don’t remember the next few days at all well; they are all a blur of looking at that one line of program over and over, trying to imagine how it could possibly be malfunctioning without causing the offending variable to be caught and printed.

I must have walked past the system programmer while waiting to myself out loud, “What could possibly be wrong? It’s acting as if the bug I fixed were still there!” “Let me have another look at your load map,” said the S.P. “Which one? I have dozens of them by now.” “The one where you recompiled with the bigger array and link-edited into your library.” I forked it over.

“Ah sol! Look here at the bottom line! It stored the program in your library under a new name, instead of replacing the old version. You’ve been running the old version all this time!”

I confessed I had never paid any attention to that bottom line. It looked just as much like gibberish as it always did. His trained eye was seeing something that wasn’t apparent to my bloodshot one. Of course it was obvious when he explained it. The linkage editor had found a program already there in the library when I had recompiled, so instead of writing over the old one with the new it had stored the new one under a different name of its own manufacture.

“Your control cards say ‘DELETE DELET’ on the control card. The first DELETE is what you want it to do if the job terminates normally, and the second one is optional and allows you to specify the action to be taken if the job terminates abnormally.

I had forgotten about that. Brand Y, in a similar situation, would have typed DUP LIBRARY on the console typewriter and waited for the operator to tell him whether to kill the job or to kill the old files and proceed.

“Now remember, if you decide to do this in the future you have to say ‘DELETE, DELETE’ on the control card. The first DELETE is what you want it to do if the job terminates normally, and the second one is optional and allows you to specify the action to be taken if the job terminates abnormally.

But for now you will have to get the operator to delete the data sets and run the job again.”

Dear Reader, if you have followed me this far you will be glad to hear that on the next run the job did run to normal termination at 53 minutes. I discovered from the output that, shortly following the case that had caused the invalid array index earlier, there was an unusually complicated set of input data which caused processing time to increase much more than I would have expected. So maybe I should be thankful that the time-limit field of the job card hasn’t worked for the past several years.

Brand X is a very highly respected and popular computer system. If your programs never generate invalid array indices, if you are worried that you might accidently wipe out an obsolete version of a program when you recompile, if you want to keep scratch files around after a program blows up, and above all if you delight in hexadecimal cryptoanalysis as a means of relating run-time errors to the source program, then you should join the ranks of thousands of satisfied users of Brand X.

But if you have some work you want to get done you would probably be better off with Brand Y.