Real Programmers Don't Use Fortran, Either!

Ed Nather

1983

A recent article devoted to the *macho* side of programming ("Real Programmers Don't Use Pascal," by ucbvax!G:tut) made the bald and unvarnished statement

> Real Programmers write in Fortran.

Maybe they do now, in this decadent era of Lite beer, hand calculators and "user-friendly" software, but back in the Good Old Days, when the term "software" sounded funny and Real Computers were made out of drums and vacuum tubes, Real Programmers wrote in machine code. Not Fortran. Not RATFOR. Not, even, assembly language. Machine Code. Raw, unadorned, inscrutable hexadecimal numbers. Directly.

Lest a whole new generation of programmers grow up in ignorance of this glorious past, I feel duty-bound to describe, as best I can through the generation gap, how a Real Programmer wrote code. I'll call him Mel, because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp., a now-defunct subsidiary of the typewriter company. The firm manufactured the LGP-30, a small, cheap (by the standards of the day) drum-memory computer, and had just started to manufacture the RPC-4000, a much-improved, bigger, better, faster — drum-memory computer. Cores cost too much, and weren't here to stay, anyway. (That's why you haven't heard of the company, or the computer.)

I had been hired to write a Fortran compiler for this new marvel and Mel was my guide to its wonders. Mel didn't approve of compilers.

"If a program can't rewrite its own code," he asked, "what good is it?"

Mel had written, in hexadecimal, the most popular computer program the company owned. It ran on the LGP-30 and played blackjack with potential customers at computer shows. Its effect was always dramatic. The LGP-30 booth was packed at every show, and the IBM salesmen stood around talking to each other. Whether or not this actually sold computers was a question we never discussed.

Mel's job was to re-write the blackjack program for the RPC-4000. (Port? What does that mean?) The new computer had a one-plus-one addressing scheme, in which each machine instruction, in addition to the operation code and the address of the needed operand, had a second address that indicated where, on the revolving drum, the next instruction was located. In modern parlance, every single instruction was followed by a GO TO! Put *that* in Pascal's pipe and smoke it.

Mel loved the RPC-4000 because he could optimize his code: that is, locate instructions on the drum so that just as one finished its job, the next would be just arriving at the "read head" and available for immediate execution. There was a program to do that job, an "optimizing assembler," but Mel refused to use it.

"You never know where it's going to put things," he explained, "so you'd have to use separate constants."

It was a long time before I understood that remark. Since Mel knew the numerical value of every operation code, and assigned his own drum addresses, every instruction he wrote could also be considered a numerical constant. He could pick up an earlier "add" instruction, say, and multiply by it, if it had the right numeric value. His code was not easy for someone else to modify.

I compared Mel's hand-optimized programs with the same code massaged by the optimizing assembly program, and Mel's always ran faster. That was because the "top-down" method of program design hadn't been invented yet, and Mel wouldn't have used it anyway. He wrote the innermost parts of his program loops first, so they would get first choice of the optimum address locations on the drum. The optimizing assembler wasn't smart enough to do it that way.

Mel never wrote time-delay loops, either, even when the balky Flexowriter

required a delay between output characters to work right. He just located instructions on the drum so each successive one was just *past* the read head when it was needed; the drum had to execute another complete revolution to find the next instruction. He coined an unforgettable term for this procedure. Although "optimum" is an absolute term, like "unique," it became common verbal practice to make it relative: "not quite optimum" or "less optimum" or "not very optimum." Mel called the maximum time-delay locations the "most pessimum."

After he finished the blackjack program and got it to run, ("Even the initializer is optimized," he said proudly) he got a Change Request from the sales department. The program used an elegant (optimized) random number generator to shuffle the "cards" and deal from the "deck," and some of the salesmen felt it was too fair, since sometimes the customers lost. They wanted Mel to modify the program so, at the setting of a sense switch on the console, they could change the odds and let the customer win.

Mel balked. He felt this was patently dishonest, which it was, and that it impinged on his personal integrity as a programmer, which it did, so he refused to do it. The Head Salesman talked to Mel, as did the Big Boss and, at the boss's urging, a few Fellow Programmers. Mel finally gave in and wrote the code, but he got the test backwards and, when the sense switch was turned on, the program would cheat, winning every time. Mel was delighted with this, claiming his subconscious was uncontrollably ethical, and adamantly refused to fix it.

After Mel had left the company for greener pa$ture$, the Big Boss asked me to look at the code and see if I could find the test and reverse it. Somewhat reluctantly, I agreed to look. Tracking Mel's code was a real adventure.

I have often felt that programming is an art form, whose real value can only be appreciated by another versed in the same arcane art; there are lovely gems and brilliant coups hidden from human view and admiration, sometimes forever, by the very nature of the process. You can learn a lot about an individual just by reading through his code, even in hexadecimal. Mel was, I think, an unsung genius.

Perhaps my greatest shock came when I found an innocent loop that had no test in it. No test. *None.* Common sense said it had to be a closed loop, where the program would circle, forever, endlessly. Program control passed

right through it, however, and safely out the other side. It took me two weeks to figure it out.

The RPC-4000 computer had a really modern facility called an index register. It allowed the programmer to write a program loop that used an indexed instruction inside; each time through, the number in the index register was added to the address of that instruction, so it would refer to the next datum in a series. He had only to increment the index register each time through. Mel never used it.

Instead, he would pull the instruction into a machine register, add one to its address, and store it back. He would then execute the modified instruction right from the register. The loop was written so this additional execution time was taken into account — just as this instruction finished, the next one was right under the drum's read head, ready to go. But the loop had no test in it.

The vital clue came when I noticed the index register bit, the bit that lay between the address and the operation code in the instruction word, was turned on — yet Mel never used the index register, leaving it zero all the time. When the light went on it nearly blinded me.

He had located the data he was working on near the top of memory — the largest locations the instructions could address — so, after the last datum was handled, incrementing the instruction address would make it overflow. The carry would add one to the operation code, changing it to the next one in the instruction set: a jump instruction. Sure enough, the next program instruction was in address location zero, and the program went happily on its way.

I haven't kept in touch with Mel, so I don't know if he ever gave in to the flood of change that has washed over programming techniques since those long-gone days. I like to think he didn't. In any event, I was impressed enough that I quit looking for the offending test, telling the Big Boss I couldn't find it. He didn't seem surprised. When I left the company, the blackjack program would still cheat if you turned on the right sense switch, and I think that's how it should be. I didn't feel comfortable hacking up the code of a Real Programmer.