Co-existing with on-line systems

MICHAEL A. PILLA
Bell Laboratories, Holmdel, New Jersey 07733

A typical cliche in the computer world is "Computers are dumber than people–but smarter than programmers." Unfortunately, this is even more of a truism when real-time programming is discussed. Few existing systems, commercially available, allow the nonprogrammer to concentrate on his field of interest, such as psychology. Rather, one is faced with the prospect of becoming a programmer in order to effectively use on-line systems in the lab. One is then faced with the prospect of becoming a mediocre programmer and eventually a mediocre researcher as well, since more time is spent trying to program the research rather than with the research itself.

I use the word "mediocre" because it is too easy to fall into the trap of working on the system for the sake of the machine itself and to devise elaborate experiments which are not really meaningful or necessary. At first, more experiments were generated with the use of computers, but the number of published results did not significantly increase. In effect, researchers were able to weed out weak experiments or procedures early with quick pilot runs. In the past, without computer aids, quite a bit of time and effort was expended on an idea. If the idea was not successful, it was usually deemed necessary to publish in order to justify the time spent. Thus, with computers, a trend appeared which saw better experiments being published even though the quantity remained about the same. I feel, however, that there is a growing tendency to publish the use of the lab computer in some frequently trivial application as a novelty. Such papers only dilute the effort which should be made on the particular field of interest and which should not be made on computers for their own sake. In addition to the above mentioned comments, there are pitfalls which every beginning real-time programmer must face and are usually better left to the full-time programmer.

Thus, the beginner must decide now whether he wishes to be known as a researcher in his chosen field or as a programmer. Many of the old-timers have developed elaborate on-line labs and have become quite expert programmers. But the best of these people are still amateurs compared with the crops of programmers available at any computer-science department. Furthermore, even if money were available to hire a full-time programmer to free the researcher from these tasks, one quickly finds that such programmers do not wish to make a career out of application-type work. Thus, you are faced with the prospect of high turnover and subsequent retraining or allowing a "new," "ultimate," etc., language or operating system to be generated to keep the programmers happy. Would it not be better to have this done once and for all by experts, preferably available from a manufacturer as a package?

The next few sections will discuss the typical pitfalls faced by real-time programmers, along with some comments about programming languages and trends.

SOME PITFALLS

The usual problems with using on-line systems are: priority interrupts, input/output waiting, languages and operating systems, and the usual stepchild, debugging.

Priority interrupts are somewhat unique to on-line systems. Admittedly, this is an oversimplification and does not give credit to the rather sophisticated interrupt structures on large, batch-oriented systems. Typically, however, the large machine programmer rarely, if ever, has to worry about such hardware questions, since there is an elaborate system of protection. Usually, in a conventional batch program, the program controls the outside world, e.g., READ a card, PRINT a line. In the equation "A = B + C," B and C are either known or are read from some device and are available for the computation. In an on-line environment, B and C may not have arrived yet (e.g., response latency, value of response), and so the computation really cannot proceed until the outside world signals the program that data are available. The data, however, may come unexpectedly and must interrupt the process which is currently in control to say, in effect, "pay attention to me, I have something important for you." Life would be rather simple and easily manageable under these circumstances, but this can be carried further. More than one such signal can occur within a given time interval and then decisions must be made about which important signal is more important than the others. Invariably, diabolical timing situations arise which are unforseen by the beginning (and sometimes expert) programmer and occasionally cause catastrophic failure of the experiment. This usually occurs after a week or two of running a paid S and causes an abort of the entire work on that S. Attempts to troubleshoot the problem result in a clean bill of health to the system, and a "glitch" is assumed–until it repeats under actual running conditions.

To demonstrate that this is not just a contrived "special situation," consider the following simple experiment. A stimulus is generated and a response is to be received subject to some given time interval. Typically, the response is on one interrupt and the time-out clock is on another. If the S responds within the interval, cancel the time-out routine, process the response, and then generate a new stimulus. On the other hand, if the S fails to respond in the interval, cancel the response acceptance routine and possibly generate some new stimulus. This trivial example has buried within itself the seeds of destruction. All

proceeds well until a response is made in the finite interval between receipt of a time-out interrupt and the cancallation of the response. It is thus possible to proceed down two different paths of the paradigm and under usual circumstances is nearly impossible to replicate. As you are all aware, when working with people, such circumstances are bound to arise for any similar situation.

The programming languages available on most on-line systems range from simple assemblers to sophisticated problem-oriented languages tailored for the specific discipline. For purposes of discussion, three categories are of interest: assembled, compiled, and interpreted.

When computers were slower than today's cycle times and compilers were not generally available, Assembly code made considerable sense for optimation of running time, size, and similar parameters. However, with the faster cycle times available today, along with minimal compilers (such as FORTRAN IV), the need for Assembly language has diminished. It is probably safe to say that for 95% of the programs written in on-line labs today, Assembly language is not really needed, for speed or space considerations. With the exception of such areas as, for example, speech processing and large display generation, the probability is high that the problem is structured wrong if Assembly code is needed. Perhaps the task being attempted is too large an increment and cannot be readily understood even if the program is successful.

Some of the commercially available interpreter-type languages are slow, admittedly, but they have advantages, usually in symbolic debugging and quickness of implementation, that can easily outweigh their timing constraints in a single-user general-purpose lab. The real problem with languages, and an associated easy pitfall, is to decry the use of what is commercially available and attempt to "re-invent the wheel" by designing a special-purpose language which usually lacks flexibility for subsequent modifications or changes in direction. If any area deserves special treatment, it has to be the area of debugging programs. Rarely do you find that a given program, written for an on-line experiment, is completely error free at the first attempt. Rather, it is commonly found that an iterative process takes place, with better understanding of both the problem and program occurring along the way. This is frustrated by the need to anticipate error spots by incorporating dumping statements or similar noninteractive error checks. Even interactive break-point analysis is not always useful if the original program is compiled into Assembly code, and the debugging program requires one to know the underlying code. It is far better to have available debugging aids which allow the user to communicate in the source language of the program, referring, for example, to the variables by name rather than by machine address. Even better is the capability of adding or modifying code at the interactive session to check out new ideas or correct fallacious reasoning. As mentioned above, interpreters usually provide this capability to varying degrees.

SOME TRENDS

Some of the newer compilers, particularly the interactive types, have attempted to correct a common programming deficiency, namely, unsupplied or insufficient code. If, for example, a variable is used before it is initialized, most present-day compilers and/or systems use the "garbage" contents. Some of the newer systems recognize this fact and request, at run time, the initial value. A logical step, and one which is being considered at various computer-science departments, is to ask for unsupplied code if a program branch is newly undertaken without initially supplied code. Interactively, code could be supplied which may hold for the remainder of the running, for just this one occasion, or perhaps a permanent record could be made of the additions.

Another area of interest which is fast showing strong potential for interactive programming is the use of microprogrammed computers, whereby the "slow" interpreter is executed in a pseudo-higher-level machine. This alleviates some of the difficulties with timing and is possible because of the cost reductions as well as speed improvements in typical machines available today.

The costs of time-sharing are rapidly decreasing, and there is a growing tendency to have small stand-alone time-sharing systems for a few users without the usual overhead such systems entail. Such systems allow program development, longer-term intermediate storage, and even interactive data analysis to take place. These features can considerably improve the lot of the individual experimenter who has limited funds and cannot afford the disk memories or industry-compatible magnetic tape units.

In line with the above, an area receiving more attention recently is the use of interactive data analysis. Quite often, the data from an experiment are handled no differently with the availability of an on-line computer than before. In fact, very little attention is paid to which data should be collected since an overwhelming amount is potentially available. Storage or media limitations may cause some inconvenience, but the actual analysis is still performed on desk calculators, or perhaps, if the problem is large enough, the analysis is programmed on a large batch computer. With natural dialogue systems, it is possible to interactively ask questions about the data such as frequency distribution, count, means, etc.; but with interactive masks and conditions (e.g., Column 26 must be greater than 0 and Columns 45-52 must be between −5 and +9). Such natural dialogue systems can be easily implemented with the help of compiler-compilers. An example of a good interactive compiler-compiler is the work reported at Brunel University, Uxbridge, England, at their September 1972 meeting (ONLINE 72) by L. E. Heindel and J. T. Roberto (LANG-PAK—an interactive incremental

compiler-compiler). With access to the many useful professionals and highly skilled graduate students in the computer-science departments, help in setting up such a data analysis system is not far away.

ERRATUM

LOVELL, J. D., NAGEL, D. C., & CARTERETTE, E. C. Digital filtering and signal processing. Behavior Research Methods and Instrumentation, 1973, 5, 21-33. Part of Subroutine RESPS on page 30 was wrong. The correct printout is below.

```
C        CONTINUOUS REAL AXIS POLES OR ZERO
100   P= CR*CA-CI*CW
      CI=CI*CA+CR*CW
      CR=P
C
C        SAMPLED REAL AXIS POLE OR ZERO
      TR=1.-EXP(-SA)*COS(SW)
      TI=    EXP(-SA)*SIN(SW)
      P= SR*TR-SI*TI
      SI=SI*TR+SR*TI
      SR=P
      RETURN
C
C        CONTINUOUS CONJUGATE POLE OR ZERO PAIR
200   TR=CA**2+CB**2-CW**2
      TI=2.*CA*CW
      P= CR*TR-CI*TI
      CI=CI*TR+CR*TI
      CR=P
C
C        SAMPLED CONJUGATE POLE OR ZERO PAIR
      TR=1.+EXP(-2.*SA)*COS(2.*SW)-2.*EXP(-SA)*COS(SB)*COS(SW)
      TI=  -EXP(-2.*SA)*SIN(2.*SW)+2.*EXP(-SA)*COS(SB)*SIN(SW)
      P= SR*TR-SI*TI
      SI=SI*TR+SR*TI
      SR=P
      RETURN
      END
      END$
```